DISEÑO Y CONTROL DE UN ROBOT REAL MEDIANTE ROS EN DIFERENTES NIVELES DE PROGRAMACIÓN

UNIVERSIDAD DE VALLADOLID ESCUELA
DE INGENIERIAS INDUSTRIALES

Grado en Ingeniería electrónica industrial y automática

Autor:

Alberto Ruiz Paris

Tutor: Alberto Herreros López

Departamento: Ingeniería de Sistemas y Automática

Valladolid, Marzo de 2023.

AGRADECIMIENTOS

En primer lugar quiero dar la gracias a mis padres y a mi hermana por apoyarme en todo momento a no rendirme y a superar todos los obstáculos que se ponían en mi camino.

Agradecer también a mis amigos y compañeros por estar en los buenos y los malos momentos.

Por último dar las gracias a mi tutor de Trabajo de Fin de Grado Alberto Herreros por dirigir mi proyecto.

RESUMEN

En el presente proyecto se explica la forma de trabajar con ROS y con todas las herramientas de las que dispone.

Dichas herramientas permiten la programación y la visualización de cualquier tipo de robot que disponga de los archivos necesarios.

En este trabajo con el fin de entender los conceptos fundamentales y de un nivel superior se va a trabajar con el robot Niryo Ned, el cual es un robot colaborativo con fines educativos que dispone la Universidad.

Palabras clave: ROS, Niryo Ned, simulación, control.

ABSTRACT

In this project are explained the different ways to work with ROS and all the tools that it have.

This tools allows the programming and visualization of any type of robot that has the necessary files.

In order to understand all the fundamental and higher level concepts is going to be used the robot called Niryo Ned, which is a collaborative robot with educational purposes that the University have.

Key words: ROS, Niryo Ned, simulation, control.

Tabla de contenido

1. Introducción ... 5
1.1 Objetivo del proyecto ... 6
2. Robot colaborativo Niryo Ned ... 7
2.1 Descripción del hardware ... 7
2.2 Descripción del software ... 9
3. ROS y entornos de programación ... 11
3.1 ROS (*Robot Operating System*) ... 11
3.1.1 Porque utilizar ROS ... 12
3.1.2 Aplicaciones reales de ROS ... 13
3.1.3 Formas de trabajar con ROS ... 14
3.1.4 Instalación de ROS ... 15
3.1.5 Conceptos básicos de ROS ... 18
3.1.6 Estructura y organización de los paquetes descargados ... 24
3.2 Gazebo ... 29
3.2.1 Creación de espacios virtuales en Gazebo ... 29
3.3 RVIZ ... 31
3.4 Moveit! ... 32
3.5 URDF (Unified Robot Description Format) ... 37
3.5.1 Xacro ... 39
3.5.2 Creación de un robot ... 39
4. Ejemplos de programación en distintos niveles ... 43
4.1 Programación con Pyniryo ... 43
4.1.1 Control básico del NiryoNed ... 44
4.1.2 Control manual mediante el mando de la Xbox360 ... 45
4.2 NiryoStudio ... 46
4.3 Comunicación entre 2 nodos atreves de un topic ... 50
4.4 Acceso a datos de un nodo. ... 52
4.5 Creación y uso de un servicio con programación mediante paquetes de Moveit y Rospy ... 55
4.5.1 Nodo “niryo_mov” ... 57

4.5.2 Nodo “niryo_menu” 61
4.5.3 Simulación con Gazebo 62
4.6 Aplicación del mismo código en 2 robots simulares. 68
5. Conclusión 69
6. Anexos 70
6.1 Anexo 1: Programa brazo_robot.urdf 70
6.2 Anexo 2: Programa de control del Niryo Ned mediante el mando de la xbox360 75
6.3 Anexo 3: Programa de nodo emisor 81
6.4 Anexo 4: Programa de nodo receptor 81
7. Bibliografía 82

ILUSTRACIÓN 1: ROBOTS COLABORATIVOS MÓVILES 5
ILUSTRACIÓN 2: ROBOT COLABORATIVOS FIJO 5
ILUSTRACIÓN 3: ROBOT COLABORATIVO NIRYO NED 7
ILUSTRACIÓN 4: ÁREA DE TRABAJO DEL NIRYO NED 8
ILUSTRACIÓN 5: ÁREA DE TRABAJO DEL NIRYO NED 8
ILUSTRACIÓN 6: RASPBERRY PI 4 8
ILUSTRACIÓN 7: ENTRADAS Y SALIDAS DEL NIRYO NED 9
ILUSTRACIÓN 8: VISIÓN GLOBAL DE LA ORGANIZACIÓN DEL SOFTWARE 10
ILUSTRACIÓN 9: LOGOTIPO ROS MELODIC 11
ILUSTRACIÓN 10: VERSIONES DE ROS EXISTENTES 12
ILUSTRACIÓN 11: ROBOT AGV 13
ILUSTRACIÓN 12: RESPUESTA QUE SE DEBE OBTENER AL AÑADIR LA CLAVE 16
ILUSTRACIÓN 13: EJEMPLO DE COMUNICACIÓN MEDIANTE MENSAJES 19
ILUSTRACIÓN 14: EJEMPLO DE COMUNICACIÓN MEDIANTE SERVICIOS 19
ILUSTRACIÓN 15: SIMULADOR TURTLESIM 20
ILUSTRACIÓN 16: TOPICS DONDE PUBLICA EL NODO TURTLESIM 21
ILUSTRACIÓN 17: TOPICS DONDE ESTÁ SUSCRITO EL NODO TURTLESIM 22
ILUSTRACIÓN 18: SERVICIOS DEL NODO TURTLESIM 22
ILUSTRACIÓN 19: APLICACIÓN DEL COMANDO ROSTOPIC INFO 22
ILUSTRACIÓN 20: APLICACIÓN DEL COMANDO ROSMSG INFO 23
ILUSTRACIÓN 21: APLICACIÓN DEL COMANDO ROSSERVICE INFO 23
ILUSTRACIÓN 22: PAQUETES DESCARGADOS DEL NIRYO NED 24
ILUSTRACIÓN 23: CARPETAS QUE FORMAN EL PAQUETE NIRYO_ROBOT_TOOLS_COMMANDER 25
ILUSTRACIÓN 24: CARPETAS QUE FORMAN EL PAQUETE NIRYO_ROBOT_DESCRIPTION 26
ILUSTRACIÓN 25: CARPETAS QUE FORMAN AL PAQUETE NIRYO_ROBOT_BRINGUP ... 27
ILUSTRACIÓN 26: ARCHIVOS .LAUNCH DISPONIBLES 27
ILUSTRACIÓN 27: CONTENIDO DEL ARCHIVO DESKTOP_GAZEBO_SIMULATION.LAUCNH 28
ILUSTRACIÓN 28: CARPETAS QUE FORMAN EL PAQUETE NIRYO_ROBOT_MSGS 29
ILUSTRACIÓN 29: LOGOTIPO DE GAZEBO 29
ILUSTRACIÓN 30: ARCHIVO NIRYO_ROBOT_GAZEBO_WORLD.LAUNCH.XML 30
ILUSTRACIÓN 31: LOGOTIPO DE RVIZ 31
ILUSTRACIÓN 32: LOGOTIPO DE MOVEIT 32
ILUSTRACIÓN 33: ESQUEMA DEL FUNCIONAMIENTO DE MOVEIT 33
ILUSTRACIÓN 34: ESQUEMA DEL FUNCIONAMIENTO DEL NODO MOVE_GROUP 34
ILUSTRACIÓN 35: TOPICS DONDE ESTÁ SUSCRITO EL NODO MOVE_GROUP 35
ILUSTRACIÓN 36: TOPICS DONDE PUBLICA EL NODO MOVE_GROUP 35
ILUSTRACIÓN 37: SERVICIOS DEL NODO MOVE_GROUP 36

ILUSTRACIÓN 38: DESCRIPCIÓN GRÁFICA DE LAS UNIONES 37
ILUSTRACIÓN 39: EJEMPLO DE LA DESCRIPCIÓN DE UN ENLACE.............................. 38
ILUSTRACIÓN 40: REPRESENTACIÓN ONLINE DE MI ROBOT 39
ILUSTRACIÓN 41: INTERFAZ DEL ASISTENTE DE MOVEIT.. 40
ILUSTRACIÓN 42: RESPUESTA DE CHECK_URDF 41
ILUSTRACIÓN 43: REPRESENTACIÓN EN MOVEIT DE MI ROBOT............................... 42
ILUSTRACIÓN 44: LOGOTIPO DE PYNIRYO.. 43
ILUSTRACIÓN 45: CÓDIGO DE MOVIMIENTOS DEL ROBOT MEDIANTE PYNIRYO......... 44
ILUSTRACIÓN 46: PICK AND PLACE MEDIANTE VISIÓN ARTIFICIAL 44
ILUSTRACIÓN 47: MANDO XBOX360.. 45
ILUSTRACIÓN 48: DATOS PROPORCIONADOS POR NIRYOSTUDIO 46
ILUSTRACIÓN 49: VISIÓN EN 3D EN TIEMPO REAL.. 47
ILUSTRACIÓN 50: CURSORES PARA EDITAR EL VALOR DE CADA JOINT 48
ILUSTRACIÓN 51: POSICIÓN DEL TCP... 48
ILUSTRACIÓN 52: MOVIMIENTO DEL TCP MEDIANTE FLECHAS 48
ILUSTRACIÓN 53: EDICIÓN EN TIEMPO REAL DE CADA JOINT 49
ILUSTRACIÓN 54: INTERFAZ DE CONFIGURACIÓN DE LAS ENTRADAS Y SALIDAS DIGITALES 49
ILUSTRACIÓN 55: PROGRAMA NODO EMISOR 51
ILUSTRACIÓN 56: PROGRAMA NODO RECEPTOR 51
ILUSTRACIÓN 57: LISTADO DE NODOS ACTIVOS 52
ILUSTRACIÓN 58: EJECUCIÓN DEL NODO RECEPTOR... 52
ILUSTRACIÓN 59: LISTADO DE NODOS ACTIVOS CON GAZEBO 53
ILUSTRACIÓN 60: MENSAJE PUBLICADO EN EL TOPIC /JOINT_STATES....................... 54
ILUSTRACIÓN 61: PROGRAMA PARA SUSCRIBIRSE AL NODO JOINTS_STATES 54
ILUSTRACIÓN 62: SETUP.PY DEL NODO MOV 56
ILUSTRACIÓN 63: SETUP.PY DEL NODO MENÚ 56
ILUSTRACIÓN 64: CONFIGURACIÓN DEL NODO MOV... 57
ILUSTRACIÓN 65: FUNCIÓN ENCARGADA DE GESTIONAR EL SERVICIO...................... 58
ILUSTRACIÓN 66: IMPORTACIONES NECESARIAS 59
ILUSTRACIÓN 67: NOMBRES DE LOS SUBGRUPOS DEL NIRYONED........................... 59
ILUSTRACIÓN 68: INICIALIZACIÓN DE LOS OBJETOS DE PYTHON 60
ILUSTRACIÓN 69: FUNCIÓN ENCARGADA DE CREAR EL ENTORNO DE SIMULACIÓN ... 60
ILUSTRACIÓN 70: CREACIÓN DEL SERVICIO EN EL NODO MENÚ 61
ILUSTRACIÓN 71: ENTORNO DE SIMULACIÓN EN GAZEBO 62
ILUSTRACIÓN 72: PANTALLA DE NODO MENÚ 62
ILUSTRACIÓN 73: EJEMPLO DE SELECCIÓN DE LA OPCIÓN 1 63
ILUSTRACIÓN 74: ROBOT EN POSICIÓN HOME... 63
ILUSTRACIÓN 75: EJEMPLO DE SELECCIÓN DE LA OPCIÓN 2.................................... 64
ILUSTRACIÓN 76: ROBOT EN POS0 64

ILUSTRACIÓN 77: ROBOT EN POS1 ... 64
ILUSTRACIÓN 78: ROBOT EN POSOBV ... 64
ILUSTRACIÓN 79: EJEMPLO DE SELECCIÓN DE LA OPCIÓN 3 ... 65
ILUSTRACIÓN 80: EJEMPLO DE SELECCIÓN DE LA OPCIÓN 4 ... 65
ILUSTRACIÓN 81: GRIPPER CERRADO ... 66
ILUSTRACIÓN 82: GRIPPER ABIERTO ... 66
ILUSTRACIÓN 83: CÓDIGO QUE SE ENCARGA DE EJECUTAR LA OPCIÓN 5 ... 66
ILUSTRACIÓN 84: VISIÓN DE LA CÁMARA EN POSOBV ... 67
ILUSTRACIÓN 85: CAPTURA DEL BLOQUE VERDE ... 67
ILUSTRACIÓN 86: BLOQUE VERDE DEPOSITADO EN EL ARMARIO ... 67
ILUSTRACIÓN 87: CÓDIGO UTILIZADO PARA EL NIRYONED ... 68
ILUSTRACIÓN 88: CÓDIGO UTILIZADO PARA EL 2º ROBOT COLABORATIVO ... 68

1. Introducción

El uso de robots colaborativos o cobots está en auge en las industrias debido a los nuevos métodos de control que han ido surgiendo y también debido a las ventajas que tienen respecto a un robot industrial.

Un robot colaborativo es capaz de realizar múltiples tareas por lo que tiene la capacidad de adaptarse a las distintas necesidades que surjan a lo largo del tiempo, además la mayoría de ellos tienen *machine learning* por lo que están dotados para responder a cualquier imprevisto. También son más ligeros por lo que la interacción con el humano es más fácil.

En la actualidad existen una gran variedad de robots colaborativos y la mayor diferencia que puede haber entre ellos es los que son capaces de desplazarse y los que están fijos en un lugar determinado, aunque la mayor característica por la que destaca un robot colaborativo es por estar constituidos por uno o varios brazos robóticos.

Ilustración 2: Robot colaborativos fijo

Ilustración 1: Robots colaborativos móviles

Los robots con capacidad de desplazamiento te dan la posibilidad de transportar materiales de un punto a otro por lo que podrían ser usados para almacenaje u organización en un entorno de trabajo de gran amplitud. Mientras que los que están fijados te ofrecen la capacidad de realizar pequeños movimientos con una gran tasa de repetitividad y a su vez con una gran precisión.

El hecho de que un robot colaborativo tenga tantas y tan distintas opciones de uso hace que surjan distintos métodos de control siendo siempre la base de cada una de estas formas la simplicidad y fiabilidad.

De esta forma surge ROS (en inglés *Robot Operating System*), este *framework* dota de un software para el desarrollo de robots y también provee de servicios estándar de un sistema operativo como puede ser el control a bajo nivel de los dispositivos e implementación de funcionalidad de uso común.

Actualmente existe ya la siguiente versión de ROS, ROS2, la cual nace con el objetivo de poder adaptar todos los cambios que han ido surgiendo a lo largo de los años en el mundo de la robótica. ROS es un *framework* centralizado hecho para unificar toda la robótica en un solo sistema, pero por diversas causas no es conveniente su uso en entornos industriales y por esta y otras razones se creó ROS2, para corregir los errores de raíz que tiene ROS y para fortalecer sus cimientos. Además, ROS2 utiliza las versiones de Python 3.0 y superiores que son las versiones donde se están lanzando nuevas librerías y funciones, mientras que ROS utiliza las versiones de Python 2 por lo que en este aspecto ROS también sale perdiendo. Actualmente existe la posibilidad de migrar proyectos de ROS a ROS2 siempre y cuando las dependencias existentes en ROS estén disponibles para ROS2 de lo contrario este proceso de migración es imposible.

En este proyecto se va utilizar ROS debido a que los paquetes están programados para este *framework.*

1.1 Objetivo del proyecto

El objetivo del proyecto reside en dar a conocer como empezar con ROS y de la capacidad que tiene para controlar un robot.

Los primeros pasos para empezar con ROS son los siguientes:

- Entender que es ROS y los usos que nos proporcionan todas las herramientas que dispone

- Saber cómo instalarlo y las distintas opciones que existen para trabajar con él.

- Conocer los métodos de comunicación que utiliza y saber como aplicarlos en nuestro beneficio.

- Aplicación de los conocimientos obtenidos
- Facilitar el uso de ROS como herramienta educativa para futuros alumnos de la universidad

De forma más global se quiere dar a entender la estructura interna que tiene ROS y como saber aprovecharla en nuestro beneficio.

2. Robot colaborativo Niryo Ned

2.1 Descripción del hardware

El robot Ned de la marca Niryo es un robot colaborativo en forma de brazo articulado de 6 ejes destinado para la educación o aprendizaje personal.

Ilustración 3: Robot colaborativo Niryo Ned

El hardware de este robot destaca por poder otorgarte las características propias de un robot de 6 ejes en cuanto a movilidad, pero con un peso muy reducido (8.9 Kg) y un rango de acción de hasta 440 mm.

Este robot es capaz de reproducir los movimientos más avanzados que hoy en día realiza un robot industrial, con una precisión de 0.5 mm y una repetitividad de 0.5 mm.

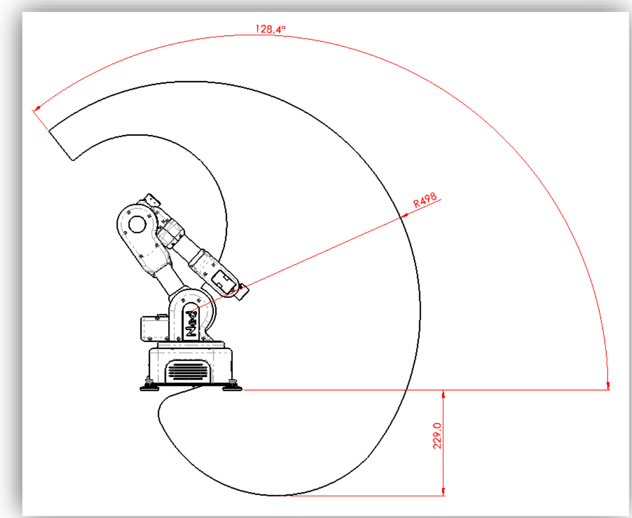

Ilustración 5: Área de trabajo del Niryo Ned

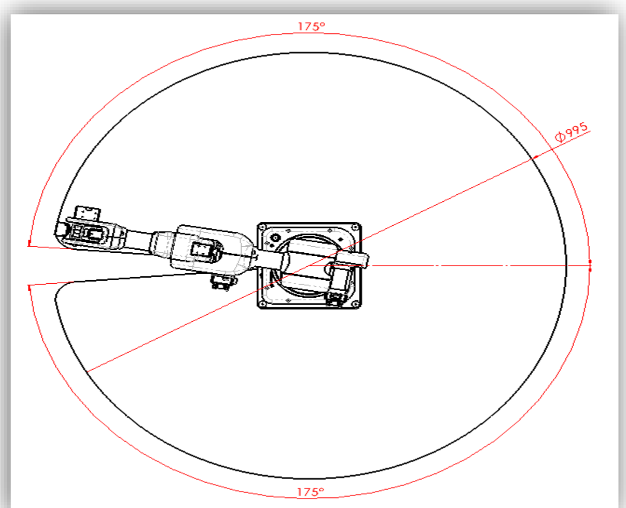

Ilustración 4: Área de trabajo del Niryo Ned

El robot Niryo Ned proporciona una interfaz eléctrica muy potente, pues contiene la Raspberry Pi 4 que va a ser la encargada de ejecutar el software y controlar las entradas y salidas digitales.

Una Raspberry Pi es un ordenador de bajo coste que consiste en una placa base que permite la conectividad de distintos componentes de un ordenador común. Todas las versiones de este miniordenador se basan tanto en hardware libre como en sistemas operativos libres basados en GNU/Linux [1].

Ilustración 6: Raspberry Pi 4

Como se puede ver en la ilustración 6 la Raspberry Pi 4 contiene 2 puertos USB 2.0, otros 2 puertos USB 3.0 y un puerto Ethernet por lo que las posibilidades de comunicación y recopilación de datos que nos permite es muy elevada y de una manera muy sencilla.

Ilustración 7: Entradas y salidas del Niryo Ned

Además, contiene un panel de conexiones donde al robot se les puede incluir hasta dos motores paso a paso a mayores, 2 conexiones de DXL – data, otras 2 conexiones de salidas digitales y 2 paneles GPIO que se pueden configurar como entradas o salidas digitales.

2.2 Descripción del software

Como hemos dicho, el “cerebro” de este robot utiliza Linux como sistema operativo.

Linux es un sistema operativo de tipo Unix que está bajo licencia GNU GPL (*General Public* o Licencia Pública General de GNU). Esto quiere decir que cualquier persona puede ejecutar el software y modificarlo de manera gratuita [2]

La siguiente imagen muestra como se organiza a nivel de comunicaciones el robot.

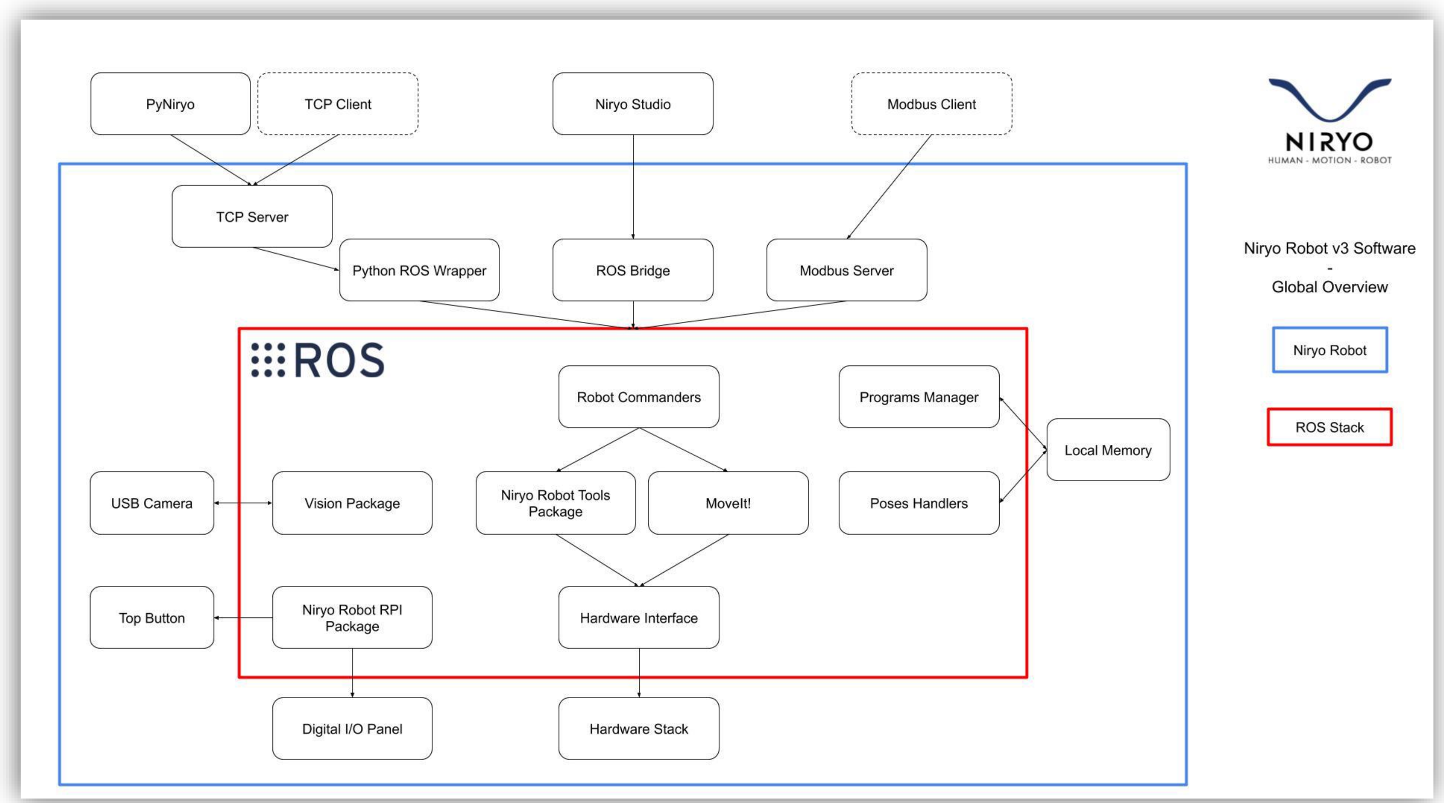

Ilustración 8: Visión global de la organización del software

En este TFG voy a explicar desde la parte interior hacia la parte superior de la ilustración 8, para que los conceptos más básicos y complejos que se deben tener para comprender la parte del recuadro rojo nos ayuden a entender de una manera más sencilla la zona superior.

Para programar el software o ejecutar simples movimientos con el robot existen diferentes métodos. En este TFG voy a centrarme en explicar como programarlo mediante ROS, PyNiryo y NiryoStudio y mostraré las diferencias que hay entre ellos en cuanto a nivel de dificultad de programación.

3. ROS y entornos de programación

3.1 ROS (*Robot Operating System*)

ROS es un entorno de trabajo que tiene como objetivo el poder desarrollar el software de un robot con la ayuda de drivers y librerías que facilitan la comunicación con el robot [3].

ROS tiene dos partes básicas:

- Ros: Es la parte que actúa cómo sistema operativo. Es donde está incluido el nodo principal de coordinación de datos, multiplexado de información y creación y destrucción de nodos

- Ros-pkg: Es el grupo de paquetes creado por el usuario y que implementan funcionalidades como bibliotecas, ejecutables, configuraciones, datos y otros recursos relacionados.

La versión de ROS que se ha utilizado es la llamada Melodic, esta versión de ROS solo es compatible con Ubuntu 18.04.

Ilustración 9: Logotipo ROS MELODIC

Cada versión de ROS que ha sido lanzada solo ha sido compatible con otra versión correspondiente de Ubuntu. De tal forma que si aparecía una nueva versión de Ubuntu, un poco más tarde se preparaba una nueva versión de ROS.

Versión	Fecha
ROS Noetic Ninjemys (Recommended)	May 23rd, 2020
ROS Melodic Morenia	May 23rd, 2018
ROS Lunar Loggerhead	May 23rd, 2017
ROS Kinetic Kame	May 23rd, 2016
ROS Jade Turtle	May 23rd, 2015
ROS Indigo Igloo	July 22nd, 2014

Ilustración 10: Versiones de ROS existentes

3.1.1 Porque utilizar ROS

ROS al ser un software de código abierto cuenta con una amplia cantidad de desarrolladores que contribuyen a mejorar el sistema. Con su variada gama de herramientas te facilita poder probar el comportamiento del robot ya sea para un laboratorio o para un entorno industrial. El poder hacer pruebas realistas mediante los simuladores antes incluso de tener el propio robot físico es una gran ventaja

puesto que te permite prever posibles fallos o incluso ahorrarte el hacer compras innecesarias puesto que antes de realizar la compra ya los has probado.

3.1.2 Aplicaciones reales de ROS

Como se comentó en la introducción, actualmente el uso de ROS en las industrias vienes a través de ROS2. El mayor uso que se da a ROS2 es para los robots tipo AGV (*Automated Guide Vehicle).*

Ilustración 11: Robot AGV

Este tipo de robots funcionan mediante unas bandas electromagnéticas colocadas en el suelo que permiten la conducción de forma autónoma de los robots creando un circuito.

La forma en la que interviene ROS en estos robots es la comunicación y protocolos que se crean para que el tráfico sea fluido y no haya colisiones o atascos o también hacer que los robots vayan a un área determinada a recargar sus baterías.

ROS también es una herramienta muy utilizada en los departamentos de I+D de muchas empresas o en entornos académicos o de investigación debido a su capacidad de simulación de entornos reales que permiten desarrollar una aplicación robótica para cualquier tipo de finalidad.

3.1.3 Formas de trabajar con ROS

Debido a que para utilizar ROS es necesario utilizar Ubuntu, se ha tenido que buscar distintas alternativas.

3.1.3.1 Mediante una máquina virtual

Una máquina virtual es un software que simula un sistema operativo como si fuera una computadora real. En mi caso he utilizado VMware, la cual permite personalizar la cantidad de recursos que vamos a utilizar del ordenador físico [4].

En cuanto a rendimiento y comodidad es la mejor opción, debido a que vas a poder ejecutar programas que requieres una gran capacidad computacional y la vez vas a poder utilizar el SO que usas normalmente y de esta forma disponer de toda la información que se haya recopilado sin tener que moverla del ordenador físico a la máquina virtual .

3.1.3.2 Mediante WSL2 de Windows

WSL2 permite ejecutar Linux en Windows a través de la ventana de comandos de Windows. Esta opción es muy parecida a la opción de la máquina virtual, pero sin la posibilidad personalizar los recursos a utilizar [5].

Esta opción no me ha sido viable utilizar debido a que los programas de simulación de robots requieren de mucha potencia computacional y mediante este método no se ejecutaban de la forma correcta.

3.1.3.3 Mediante el uso directo del sistema operativo de Ubuntu 18.04

Este método requiere tener una buena adaptación al SO de Linux. Además la instalación inicial del sistema no es sencilla y requiere de tener un conocimiento avanzado sobre el ordenador que tienes y de las configuraciones necesarias para poder aprovechar todos los recursos del ordenador.

Una vez instalado y listo para trabajar considero que es una muy buena opción para trabajar con ROS ya que el ordenador va a poder dedicarse solamente a ejecutar los programas sin tener otro sistema operativo que mantener.

3.1.4 Instalación de ROS

Al estar trabajando en Ubuntu la instalación de estos paquetes se realizara con comandos en la terminal [6].

Iré explicando los comandos más importantes y en algunos casos el por qué en necesario aplicarlos.

sudo apt-get update

Este comando actualizará la lista de paquetes del repositorio de nuestro software de distribución, este caso Ubuntu 18.04.

mkdir -p catkin_ws_niryo_ned/src

Mediante este comando estamos creando 2 carpetas, la principal la cual hemos llamado “catkin_ws_niryo_ned” y la carpeta “src” cuyo nombre proviene de *source*. En src será donde estén todos los paquetes que instalaremos posteriormente.

cd catkin_ws_niryo_ned

sudo apt install git

Instalamos “git” ya que de esta manera podremos descargar los paquetes de “github” del Niryo ned directamente en la carpeta “src” mediante el siguiente comando.

git clone https://github.com/NiryoRobotics/ned_ros src

Hasta ahora lo que se ha estado realizando es preparar el espacio de trabajo de ROS pero aún no lo tenemos instalado, para ello y antes de poner el comando que instalará

la versión de ROS que se necesita, hay que introducir otros comandos que van a permitir la instalación.

sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'

Este comando prepara al SO para aceptar paquetes de ROS.

sudo apt install curl

curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -

```
ubuntu@ubuntu1804:~/catkin_ws_niryo_ned$ curl -s https://raw.githubusercontent.c
om/ros/rosdistro/master/ros.asc | sudo apt-key add -
OK
```

Ilustración 12: Respuesta que se debe obtener al añadir la clave

En el primer comando es para instalar "curl" (*Client URL*) el cual nos permite intercambiar información obtenidas de una URL en una terminal.

Y en la segunda línea se obtiene la llave para que con la URL introducida podamos instalar ROS.

sudo apt install ros-melodic-desktop-full

Existen varios comandos para instalar ROS Melodic pero el recomendado y el que más opciones tiene para desarrollar nuestro robot es este, ya que con él también estamos instalando simuladores como Rviz y Gazebo y también librerías genéricas de robots.

Ahora los comandos que vamos a introducir son paquetes extras necesarios para el funcionamiento del Niryo Ned.

sudo apt install build-essential

sudo apt install python-catkin-pkg

sudo apt install python-pymodbus

sudo apt install python-rosdistro

sudo apt install python-rospkg

sudo apt install python-rosdep-modules

sudo apt install python-rosinstall python-rosinstall-generator

sudo apt install python-wstool

sudo apt install ros-melodic-moveit

sudo apt install ros-melodic-tf2-web-republisher

sudo apt install ros-melodic-rosbridge-server

sudo apt install ros-melodic-joint-state-publisher-gui

sudo apt install ros-melodic-joint-trajectory-controller

source /opt/ros/melodic/setup.bash

catkin_make

Para entender bien el porque de este comando primero voy a explicar que es catkin. Catkin es el sistema “build” oficial de ROS, el cual combina macros de CMake con scripts de Python para dar funcionalidad al espacio de trabajo. Con estos archivos genera código fuente que va a ser usado de forma indirecta por el usuario.

Al introducir el comando catkin_make lo que estamos haciendo es crear 2 carpetas nuevas en la misma ubicación de donde está la carpeta src. Estas dos carpetas serán la carpeta *build* donde se encuentran los archivos make y cmake y la carpeta *devel*

que contendrá cualquier archivo generado por que permitirá la ejecución de los programas que se encuentre en la carpeta src. En conclusión, al ejecutar este comando se está compilando todo el entorno de trabajo de ROS y si es necesario creará los archivos pertinentes en las carpetas *build* o *devel*

```
echo "source $(pwd)/devel/setup.bash" >> ~/.bashrc

source ~/.bashrc
```

Con estos últimos comandos añadimos a nuestra sesión bash las variables del espacio de trabajo de ROS.

Una vez introducidos estos comandos ya podremos usar ROS con el robot Niryo Ned o para cualquier otra aplicación que se quiera desarrollar.

3.1.5 Conceptos básicos de ROS

La comunicación en ROS se realiza mediante nodos. Los nodos son códigos ejecutables que pueden residir en el PC o en propio robot, cada nodo puede controlar un aspecto del sistema por lo que el control de un robot estará formado por una elevada cantidad de nodos [7]

La comunicación entre nodos se realiza mediante mensajes, los cuales son estructuras de datos simples (enteros, dobles, booleanos…) o complejos (listas, estructuras…). [8]

Los nodos envían los mensajes publicándolos en un topic. Estos topics identifican el contenido del mensaje. [9]

Si un nodo está interesado en tener cierta información que se encuentra almacenada en un mensaje, ese nodo se tendrá que suscribir al topic donde otro nodo estará publicando en el mismo topic el mensaje.

Este método de basado en la publicación y subscripción permite una gran flexibilidad, pero puede llegar un momento donde haya un exceso de datos innecesarios publicándose en un instante determinado. Para evitar esta sobrecarga se utilizan los servicios. Los servicios permiten ejecutar una interacción de petición/entrega donde un nodo solo va a publicar mediante el servicio un mensaje al nodo que lo haya solicitado. [10]

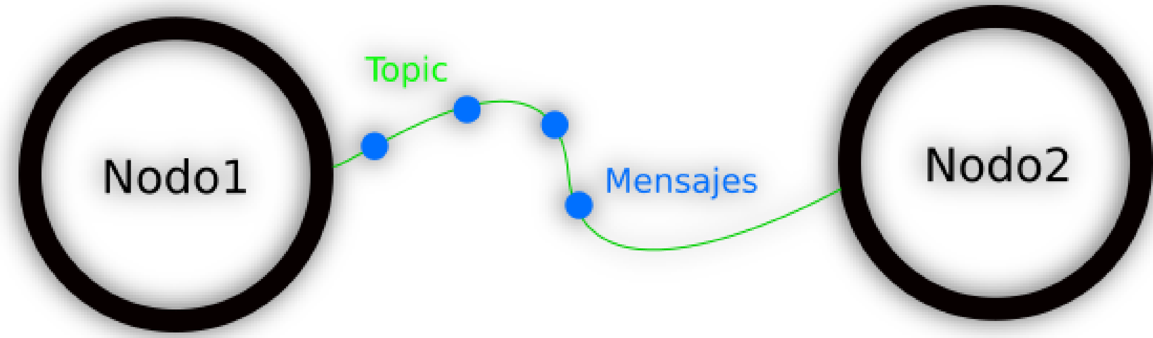

Ilustración 13: Ejemplo de comunicación mediante mensajes

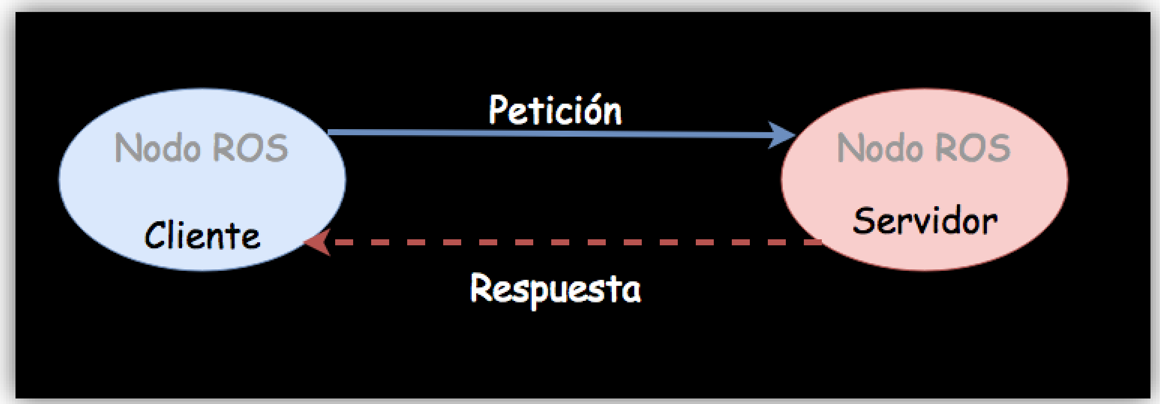

Ilustración 14: Ejemplo de comunicación mediante servicios

3.1.5.1 Aplicaciones prácticas de los conceptos

Para mostrar los conceptos básicos de ROS de una forma simple y visual me voy a ayudar de la herramienta de Turtlesim. Turtlesim es una herramienta didáctica con el objetivo de poder enseñar de una manera intuitiva las comunicaciones realizadas en un espacio de trabajo de ROS.

Si ya tenemos instalado ROS, para ejecutar Turtlesim no es más que ejecutar los dos siguientes comandos:

roscore

rosrun turtlesim turtlesim_node

Con el comando de **roscore** lo que estamos haciendo es crear una colección de nodos y programas necesarios para ejecutar ROS. Es un requisito obligatorio y que en todo momento va a tener que estar activo ya que de lo contrario el espacio de trabajo no tendrá comunicación.

Con el segundo comando aparece la siguiente pantalla:

Ilustración 15: Simulador Turtlesim

Como se puede observar ya tenemos a nuestra tortuga, la cual vamos a tener como objetivo interactuar con ella mediante los comandos básicos que nos proporciona ROS:

Comandos para los nodos:

- rosnode list: Este comando nos mostrará por la terminal todos los nodos activos que hay en ese instante
- rosnode info: Al ejecutar este comando aparecerá en la terminal información relacionada con el nodo indicado.

Comandos para los topics:

- rostopic list: Muestra todos los topics a los que se puede acceder

- rostopic info: Enseña atreves de la terminal la información más importante del topic
- rostopic echo: Muestra los mensajes que se están publicando en ese topic
- rostopic pub: Publicas datos en el topic indicado

<u>Comandos para servicios:</u>

- rosservice list
- rosservice info
- rosservice call: Ejecuta el servicio indicado con unos determinados argumentos

<u>Comandos para mensajes:</u>

- rosmsg show: Muestra los campos de datos del mensaje indicado
- rosmsg info

En esta lista de comandos he escrito los comandos que más he utilizado, pero hay más cuya utilidad puede aumentar para otros casos.

Una vez ya descritos los comandos que más vamos a utilizar comenzamos ejecutando ***rosnode list*** y obtenemos por la terminal que hay 2 nodos activos, uno de ellos es un nodo creado por el roscore, ese nodo es el rosout y el otro nodo es el que hemos ejecutado después del comando del rocore.

Ahora voy a ejecutar el comando ***rosnode info /turtlesim*** para saber de la situación de la que partimos para poder ir comparando con los escenarios que vayamos teniendo más adelante.

De la terminal obtenemos la siguiente información:

```
Node [/turtlesim]
Publications:
 * /rosout [rosgraph_msgs/Log]
 * /turtle1/color_sensor [turtlesim/Color]
 * /turtle1/pose [turtlesim/Pose]
```

Ilustración 16: Topics donde publica el nodo turtlesim

```
Subscriptions:
 * /turtle1/cmd_vel [unknown type]
```

Ilustración 17: Topics donde está suscrito el nodo turtlesim

```
Services:
 * /clear
 * /kill
 * /reset
 * /spawn
 * /turtle1/set_pen
 * /turtle1/teleport_absolute
 * /turtle1/teleport_relative
 * /turtlesim/get_loggers
 * /turtlesim/set_logger_level
```

Ilustración 18: Servicios del nodo turtlesim

Lo que nos quieren decir estas imágenes es que el nodo turtlesim está publicando en los topics *rosout, turtle1/color_sensor y turtle1/pose,* que está suscrito al topic turtle1 y que puede ejecutar hasta 9 servicios.

Para comprobar que esto es cierto vamos a ejecutar el comando ***rostopic info /turtle1/color_sensor*** y obtenemos lo siguiente:

```
ubuntu@ubuntu1804:~$ rostopic info /turtle1/color_sensor
Type: turtlesim/Color

Publishers:
 * /turtlesim (http://ubuntu1804:43559/)

Subscribers: None
```

Ilustración 19: Aplicación del comando rostopic info

Lo primero que nos indica el "*type",* que en este caso, es lo mismo que el mensaje y los segundo es el publicador que, como es lógico coincide con la información obtenida en las anteriores pantallas.

Para finalizar con esta la parte de como usar los comandos escribiremos el comando ***rosmsg info tutlesim/Color*** para ver los campos de datos que tiene ese tipo de mensaje.

```
ubuntu@ubuntu1804:~$ rosmsg info turtlesim/Color
uint8 r
uint8 g
uint8 b
```

Ilustración 20: Aplicación del comando rosmsg info

Para saber donde tenemos ubicada la tortuga utilizamos el comando ***rostopic echo /turtle1/pose.***

Si queremos mover la tortuga podemos utilizar 2 de los servicios que obtuvimos al buscar la información relacionada con el nodo turtlesim. Por ejemplo, voy a utilizar el servicio de */turtle1/teleport_absolute*. Para saber los argumentos que necesito rellenar para conseguir ejecutarlo ejecutamos el comando ***rosservice info /turtle1/teleport_absolute*** y obtenemos la siguiente respuesta:

```
ubuntu@ubuntu1804:~$ rosservice info /turtle1/teleport_absolute
Node: /turtlesim
URI: rosrpc://ubuntu1804:53861
Type: turtlesim/TeleportAbsolute
Args: x y theta
```

Ilustración 21: Aplicación del comando rosservice info

Podemos ver que para ejecutar el servicio se necesitan 3 argumentos (x, y, theta). Entonces con estos datos ya podemos ejecutar nuestro primer movimiento utilizando el comando ***rosservice call /turtle1/teleport_absolute 2 2 0***

3.1.6 Estructura y organización de los paquetes descargados

Una vez preparado nuestro entorno de ROS en la carpeta "src" se encontraran todos los archivos ejecutables que necesitara el software para su correcto funcionamiento.

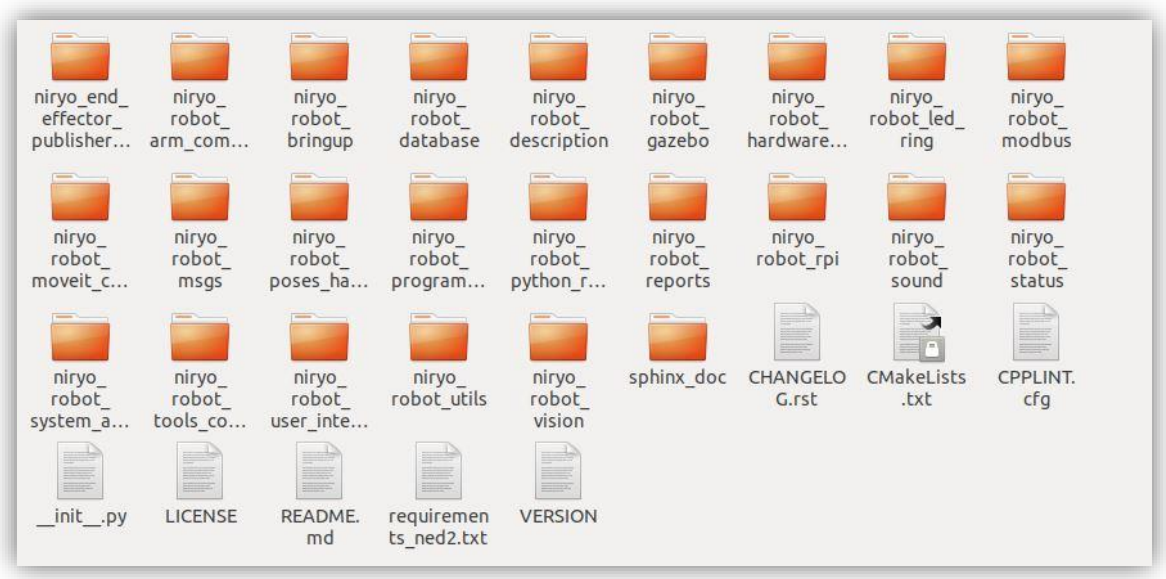

Ilustración 22: Paquetes descargados del Niryo Ned

En este apartado se va a explicar los paquetes más importantes y el cómo están estructurados en su interior. De esta forma si se consigue entender la organización interna que tiene cada paquete se podrá entender de una forma más global el comportamiento de ROS.

De entre todos los paquetes que se muestran en la ilustración 11 se pueden diferenciar entre 2 tipos:

- Los paquetes que van a ser nodos y que se encargaran de proporcionar y gestionar los datos mientras el robot se está ejecutando
  Por ejemplo, en el paquete *niryo_robot_tools_commander* está la carpeta *scripts* y en su interior hay un archivo de Python con el nombre *tool_commander_node.py*. Este archivo Python es el ejecutable que se va a encargar de hacer funcionar este nodo.

Estos archivos Python que ejecutan los nodos deben estar siempre dentro de la carpeta *scripts* por lo que si un paquete contiene esta carpeta puede que sea un nodo.

- Los que no tienen la carpeta “scripts” van a ser normalmente paquetes que van a almacenar datos sobre las características físicas del robot como puede el paquete *niryo_robot_description,* los que van a encargarse de lanzar los programas de simulación de la manera adecuada como es el caso de la carpeta *niryo_robot_bringup* o que contengan datos o programas complementarios al resto de nodos y que facilitarán las cosas como puede ser la carpeta niryo*_robot_msgs.*

Con el fin de saber interpretar que información o que objetivo cumple cada paquete voy a hacer una descripción sobre los paquetes mencionados anteriormente:

niryo_robot_tools_commander

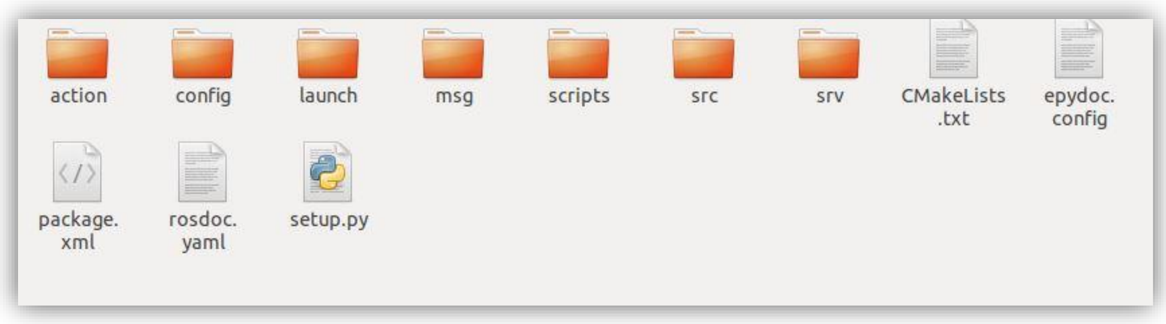

Ilustración 23: Carpetas que forman el paquete niryo_robot_tools_commander

Como se puede ver el paquete está formado por varias carpetas y algunos archivos. Los archivos son necesarios para crear el entorno de trabajo de ROS cuando se ejecuta el comando ***catkin_make.***

Las carpetas que hay tienen distintas finalidades las cuales van a ser comunes para el resto de paquetes:

- La carpeta *action* contiene un archivo .action el cual se encarga de describir el destino, un resultado y una información de retorno. Este tipo de archivos se utilizan de una forma muy parecida a los servicios.

- La carpeta “config” existe debido a que los paquetes descargados son capaces de simular tanto el robot Niryo Ned, como el Ned2 o por último el One. Entonces en función del parámetro que se asigne se ejecutara un robot u otro.

- La carpeta *launch* contiene el archivo .launch el cual permite lanzar solamente este nodo. Para poder utilizar estos archivos hay que utilizar el comando **roslaunch <nombre del paquete> <nombre del archivo a ejecutar>**

- La carpeta “msg” almacena 2 tipos de mensajes propios de este nodo

- La carpeta *scripts* se debe utilizar para almacenar los archivos que puedan ser ejecutados por los usuarios como es el caso de un nodo.

- La carpeta “src” almacena los archivos como clases u objetos.

- La carpeta “srv” almacenará los datos necesarios para crear un servicio

niryo_robot_description

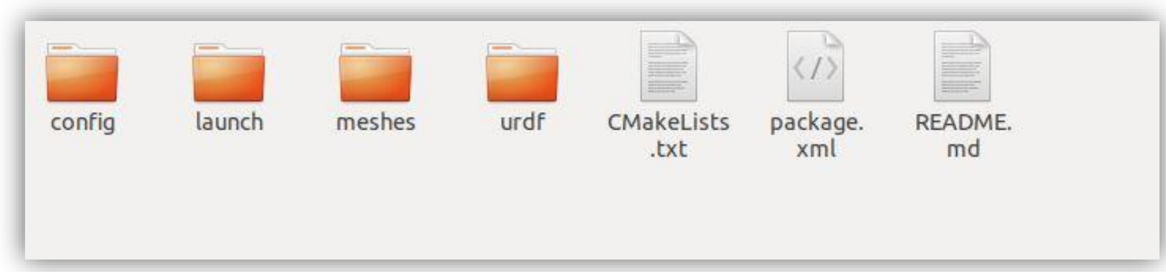

Ilustración 24: Carpetas que forman el paquete niryo_robot_description

- La carpeta *meshes* contiene los archivos .stl y .dae que serán utilizados para dar una forma más detallada al robot.

- La carpeta “urdf” contiene todos los archivos .urdf que describen al robot.

niryo_robot_bringup

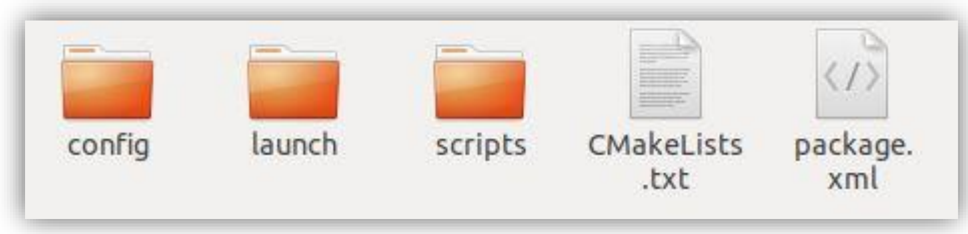

Ilustración 25: Carpetas que forman al paquete niryo_robot_bringup

En este paquete la carpeta *launch* es la más importante debido a que es la que contiene todos los archivos .launch que se van a utilizar para simular con Gazebo o Rviz.

Ilustración 26: Archivos .launch disponibles

Al ejecutar cualquiera de estos archivos lo que ocurrirá es que se empezarán a crear todos los nodos necesarios para que la simulación sea exitosa.

Por ejemplo para lanzar Gazebo sería de la siguiente manera: **roslaunch niryo_robot_bringup desktop_gazebo_simulation.launch**

Para tener claro que es lo que está haciendo este comando voy a explicar este ejemplo para que si en el futuro queremos realizar alguna modificación en estos archivos se sepa que se debe cambiar.

Con este comando se está utilizando roslaunch en el paquete *niryo_robot_bringup* y dentro de este paquete buscará ejecutar el archivo *desktop_gazebo_simulation.launch.*

```
<?xml version="1.0" ?>
<!-- This launch file allows to run simulation on Gazebo
Warning : Some errors are present in Gazebo
-> It will log some errors because it cannot unload joint_controller (It's because ROS does it before Gazebo)
-> Gazebo GUI is kinda bad on closing. It will always run into SIGTERM. So, what you should do is
to change _TIMEOUT_SIGINT value in the file /opt/ros/melodic/lib/python2.7/dist-packages/roslaunch/nodeprocess.py
This will reduce the amount of time spent waiting during program shutdown
-->
<launch xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="https://gist.githubusercontent.com/nalt/dfa2abc9d2e3ae4feb82ca5608090387/raw/roslaunch.xsd">
    <env name="ROSCONSOLE_CONFIG_FILE"
        value="$(find niryo_robot_bringup)/config/desktop_gazebo_simulation_trace.conf" />
    <!-- Arguments definition -->
    <arg name="log_level" default="INFO" />
    <arg name="hardware_version" default="ned" />
    <arg name="debug" default="false"/>
    <arg name="gripper_n_camera" default="true"/>
    <arg name="simu_conveyor" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="conf_location" default="version.txt"/>

    <!-- Define most of ROS parameters -->
    <include file="$(find niryo_robot_bringup)/launch/niryo_$(arg hardware_version)_simulation.launch">
        <arg name="simu_gripper" value="$(arg gripper_n_camera)"/>
        <arg name="simu_conveyor" value="$(arg simu_conveyor)"/>
        <arg name="log_level" value="$(arg log_level)" />
        <arg name="debug" value="$(arg debug)"/>
        <arg name="gazebo" value="true" />
    </include>

    <!-- Launching Gazebo World -->
    <include file="$(find niryo_robot_gazebo)/launch/niryo_robot_gazebo_world.launch.xml">
        <arg name="gui" value="$(arg gui)"/>
    </include>

    <param name="/niryo_robot_vision/initialized" type="bool" value="true" unless="$(arg gripper_n_camera)"/>

</launch>
```

Ilustración 27: Contenido del archivo desktop_gazebo_simulation.laucnh

En la ilustración 16 se muestra el archivo que se va a ejecutar, se puede ver que accede a distintas carpetas para así poder configurar los argumentos de la forma indicada. Para ejecutar Gazebo busca en el paquete *niryo_robot_gazebo* el archivo *niryo_robot_gazebo_world.laucnh.xml* que es el archivo que se encarga de ejecutar Gazebo con el mundo indicado.

niryo_robot_msgs

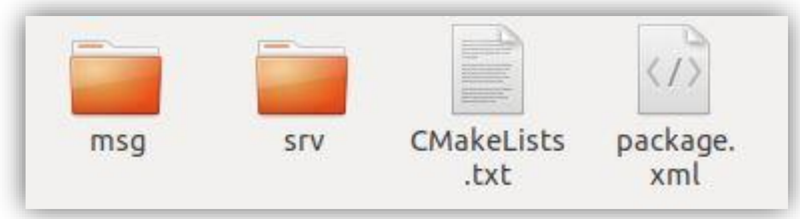

Ilustración 28: Carpetas que forman el paquete niryo_robot_msgs

Esta carpeta tiene como objetivo crear mensajes y servicios que facilitarán la programación y comunicación entre nodos.

3.2 Gazebo

Ilustración 29: Logotipo de Gazebo

Gazebo es un simulador 3D dinámico con la capacidad de simular robots con gran precisión en entornos creados por el usuario con el objetivo de que las simulaciones se ajusten lo máximo a la realidad. [11]

Además, permite la sincronización con ROS de forma que los programas que realicemos se puedan ver a través de él.

3.2.1 Creación de espacios virtuales en Gazebo

Para poder editar el espacio virtual donde se encuentra el robot debemos primero de que manera Gazebo es capaz de generar un mundo de una determinada manera.

Uno de los paquetes descargados tiene el nombre de *niryo_robot_gazebo* en su interior a parte del .txt y del .xml hay 3 carpetas que se deben destacar:

- Launch: En esta carpeta se encuentra el archivo *niryo_robot_gazebo_world.launch.xml.*

```
<?xml version="1.0" ?>
<launch xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://
www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation="https://gist.githubusercontent.com/
nalt/dfa2abc9d2e3ae4feb82ca5608090387/raw/roslaunch.xsd">
    <!-- Arguments -->
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>

    <!-- We resume the logic in empty_world.launch, changing only the name
of the world to be launched -->
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find niryo_robot_gazebo)/worlds/
niryo_cube_world.world"/>
        <arg name="debug" value="$(arg debug)"/>
        <arg name="gui" value="$(arg gui)"/>
        <arg name="use_sim_time" value="$(arg use_sim_time)"/>
        <arg name="headless" value="$(arg headless)"/>
        <arg name="output" value="log"/>
    </include>

    <!-- Spawn URDF in Gazebo -->
    <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
respawn="false"
          args="-urdf -model niryo_robot -param robot_description"
output="log"/>
</launch>
```

Ilustración 30: Archivo niryo_robot_gazebo_world.launch.xml

Dentro del archivo hay varios parámetros que son configurables. El que más interesa es el "world_name" el cual indica la ubicación del archivo donde se encuentra el mundo que queremos ejecutar.

- Models: Esta carpeta contiene los archivos necesarios para Gazebo pueda simular objetos.
  Para generar estos archivos hay que hacerlo desde el propio programa accediendo a la opción de edit y posteriormente a la opción de model editor.

Al hacer click se entrará en modo editor de Gazebo en el que se podrá añadir objetos que proporcione el programa o que se haya creado con un programa de modelado.
Una vez hayamos añadido el objeto al mundo existe la opción de variar el tamaña y propiedades físicas como la inercia o por ejemplo si le afecta la gravedad. Estas opciones son muy importantes que estén bien completadas ya que una mala configuración afectaría de forma muy negativa al desarrollo de la simulación.

- Worlds: en esta carpeta se almacenan los mundos que se han guardado de Gazebo

3.3 RVIZ

Ilustración 31: Logotipo de Rviz

Rviz es una herramienta de visualización 3D para aplicaciones de ROS de robots, sensores y algoritmos. El objetivo de Rviz es el poder permitirte visualizar el estado del robot. [12]

Una de las virtudes que hace destacar a Rviz por encima de Gazebo es el poder atribuirle complementos como cámaras, sensores *lidar* o un acelerómetro de una forma más sencilla y rápida. Mientras que su poder de simulación en entornos reales no es tan potente como Gazebo.

Es uso de una de estas herramientas depende del objetivo o el uso que vayas a querer dar a tu robot.

3.4 Moveit!

Ilustración 32: Logotipo de Moveit

Moveit es un software de código abierto para ROS que incluye diferentes características que facilitan la programación de brazos de robot y manipuladores móviles. [13]

En la siguiente ilustración 31 se puede apreciar como Moveit divide las funcionalidades en distintos bloques, siendo el más importante el bloque Move Group. Cada bloque es un conjunto de programas que todos juntos consiguen hacer más sencilla la programación del robot y su simulación.

Estos programas ejecutarán nodos que estarán suscritos y también publicarán datos mediante topics o servicios. De esta manera se genera un flujo de datos controlados y de una forma organizada y visual para que el usuario sea capaz de identificar sin mucha complicación la función que tiene cada nodo y así poder acceder a los datos que le interesen.

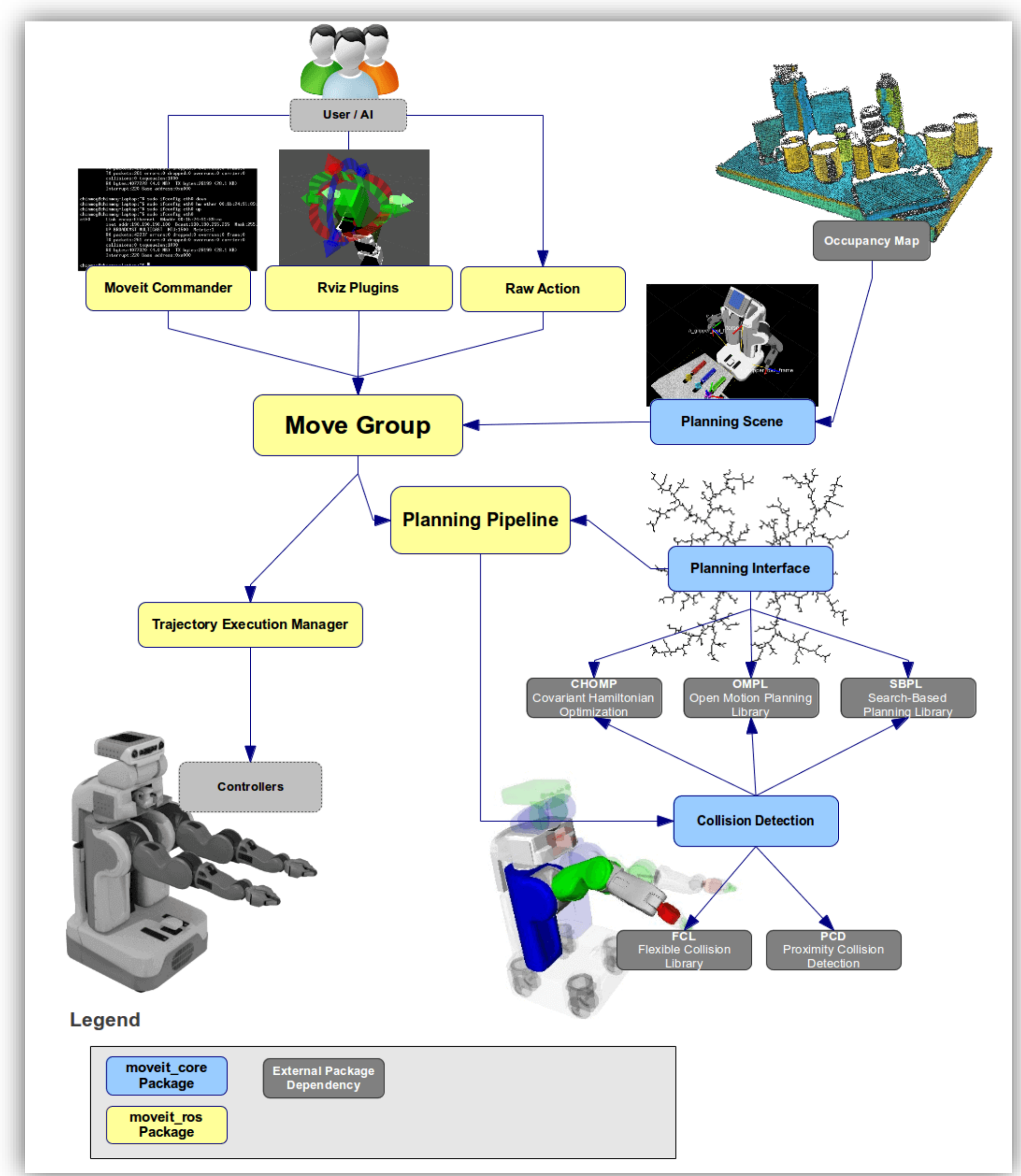

Ilustración 33: Esquema del funcionamiento de Moveit

Dentro del bloque Move Group es donde se genera el nodo move_group el cual va a tener la siguiente configuración:

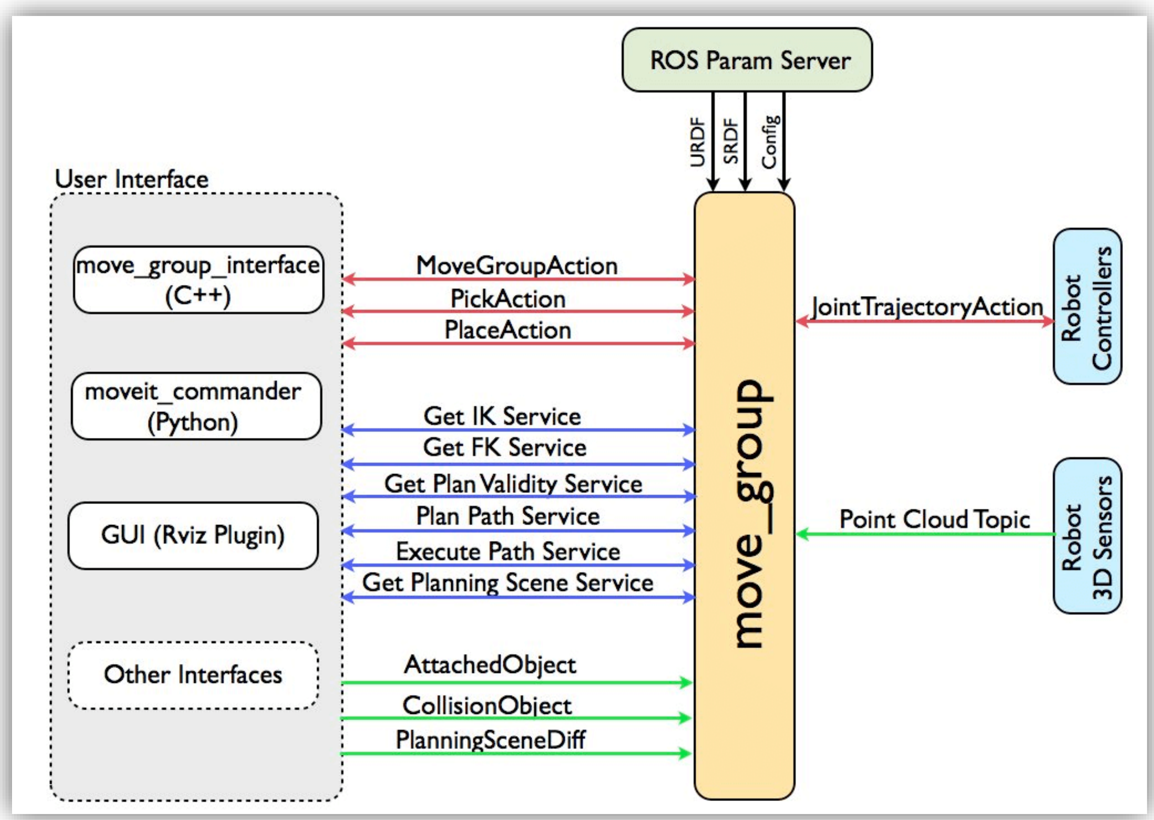

Ilustración 34: Esquema del funcionamiento del nodo move_group

Al usar Moveit este nodo es el que se va a encargar que lo que el usuario haya programado se ejecute correctamente.

Por ejemplo al ejecutar Rviz con nuestro robot Niryo Ned podemos ver la siguiente información en cuanto al nodo move_group.

```
Publications:
 * /execute_trajectory/feedback [moveit_msgs/ExecuteTrajectoryActionFeedback]
 * /execute_trajectory/result [moveit_msgs/ExecuteTrajectoryActionResult]
 * /execute_trajectory/status [actionlib_msgs/GoalStatusArray]
 * /move_group/display_contacts [visualization_msgs/MarkerArray]
 * /move_group/display_planned_path [moveit_msgs/DisplayTrajectory]
 * /move_group/feedback [moveit_msgs/MoveGroupActionFeedback]
 * /move_group/monitored_planning_scene [moveit_msgs/PlanningScene]
 * /move_group/ompl/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /move_group/ompl/parameter_updates [dynamic_reconfigure/Config]
 * /move_group/plan_execution/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /move_group/plan_execution/parameter_updates [dynamic_reconfigure/Config]
 * /move_group/planning_scene_monitor/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /move_group/planning_scene_monitor/parameter_updates [dynamic_reconfigure/Config]
 * /move_group/result [moveit_msgs/MoveGroupActionResult]
 * /move_group/sense_for_plan/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /move_group/sense_for_plan/parameter_updates [dynamic_reconfigure/Config]
 * /move_group/status [actionlib_msgs/GoalStatusArray]
 * /move_group/trajectory_execution/parameter_descriptions [dynamic_reconfigure/ConfigDescription]
 * /move_group/trajectory_execution/parameter_updates [dynamic_reconfigure/Config]
 * /niryo_robot_follow_joint_trajectory_controller/follow_joint_trajectory/cancel [actionlib_msgs/GoalID]
 * /niryo_robot_follow_joint_trajectory_controller/follow_joint_trajectory/goal [control_msgs/FollowJointTrajectoryActionGoal]
 * /pickup/feedback [moveit_msgs/PickupActionFeedback]
 * /pickup/result [moveit_msgs/PickupActionResult]
 * /pickup/status [actionlib_msgs/GoalStatusArray]
 * /place/feedback [moveit_msgs/PlaceActionFeedback]
 * /place/result [moveit_msgs/PlaceActionResult]
 * /place/status [actionlib_msgs/GoalStatusArray]
 * /rosout [rosgraph_msgs/Log]
```

Ilustración 36: Topics donde publica el nodo move_group

```
Subscriptions:
 * /attached_collision_object [moveit_msgs/AttachedCollisionObject]
 * /collision_object [unknown type]
 * /execute_trajectory/cancel [actionlib_msgs/GoalID]
 * /execute_trajectory/goal [moveit_msgs/ExecuteTrajectoryActionGoal]
 * /joint_states [sensor_msgs/JointState]
 * /move_group/cancel [actionlib_msgs/GoalID]
 * /move_group/goal [moveit_msgs/MoveGroupActionGoal]
 * /niryo_robot_follow_joint_trajectory_controller/follow_joint_trajectory/feedback [control_msgs/FollowJointTrajectoryActionFeedback]
 * /niryo_robot_follow_joint_trajectory_controller/follow_joint_trajectory/result [control_msgs/FollowJointTrajectoryActionResult]
 * /niryo_robot_follow_joint_trajectory_controller/follow_joint_trajectory/status [actionlib_msgs/GoalStatusArray]
 * /pickup/cancel [actionlib_msgs/GoalID]
 * /pickup/goal [moveit_msgs/PickupActionGoal]
 * /place/cancel [actionlib_msgs/GoalID]
 * /place/goal [moveit_msgs/PlaceActionGoal]
 * /planning_scene [moveit_msgs/PlanningScene]
 * /planning_scene_world [moveit_msgs/PlanningSceneWorld]
 * /tf [tf2_msgs/TFMessage]
 * /tf_static [tf2_msgs/TFMessage]
 * /trajectory_execution_event [std_msgs/String]
```

Ilustración 35: Topics donde está suscrito el nodo move_group

```
Services:
 * /apply_planning_scene
 * /check_state_validity
 * /clear_octomap
 * /compute_cartesian_path
 * /compute_fk
 * /compute_ik
 * /get_planner_params
 * /get_planning_scene
 * /move_group/get_loggers
 * /move_group/load_map
 * /move_group/ompl/set_parameters
 * /move_group/plan_execution/set_parameters
 * /move_group/planning_scene_monitor/set_parameters
 * /move_group/save_map
 * /move_group/sense_for_plan/set_parameters
 * /move_group/set_logger_level
 * /move_group/trajectory_execution/set_parameters
 * /plan_kinematic_path
 * /query_planner_interface
 * /set_planner_params
```

Ilustración 37: Servicios del nodo move_group

Mediante estos topics y servicios el nodo se comunicará con el robot para por ejemplo obtener la posición actual del robot y ejecutar una trayectoria.

El usuario se puede beneficiar de todos estos topics y servicios que se han generado de 3 formas distintas:

- Mediante programación en el lenguaje C++ usando la librería llamada “move_group_interface” que permite programar lo que el nodo move_group debe hacer
- Mediante Python, usando el paquete llamado Moveit_commander.
- Mediante Rviz con las distintas opciones que proporciona su interfaz visual

Para que este nodo pueda funcionar correctamente necesita tener a su disposición 3 tipos de datos que va a necesitar:

- URDF: En el siguiente apartado explicaré más en detalle lo que es, pero mediante este archivo el nodo sabrá la morfología del robot
- SRDF: Este archivo contiene información relacionada con el robot la cual no se encuentra en el URDF.
- Moveit configuration: Este archivo se genera a través del Moveit setup Assistant, este programa permite configurar los límites y controladores del robot mediante el archivo URDF.

3.5 URDF (Unified Robot Description Format)

URDF es un código basado en el metalenguaje XML para representar el modelo de un robot.

Lo primero que se necesita para crear un robot en saber como es el robot que queremos. En URDF el robot va a ser descrito con el formato XML atreves de enlaces y uniones. Cada unión o enlace la vas a poder ajustar de varias formas para que el modelo del robot se pueda ajustar lo máximo posible al de la realidad. [14]

URDF describe el robot como un árbol de enlaces que están conectados por uniones. Estas uniones pueden ser de varios tipos para que la descripción que se haga se pueda aproximar lo máximo posible al robot.

Tipos de uniones:

- Revolución: Movimiento de rotación con límite de giro
- Continuo: Movimiento de rotación sin límite de giro
- Prismático: Movimiento lineal con posiciones mínimas y máximas
- Fijo: Ambos enlaces están conectados de forma rígida

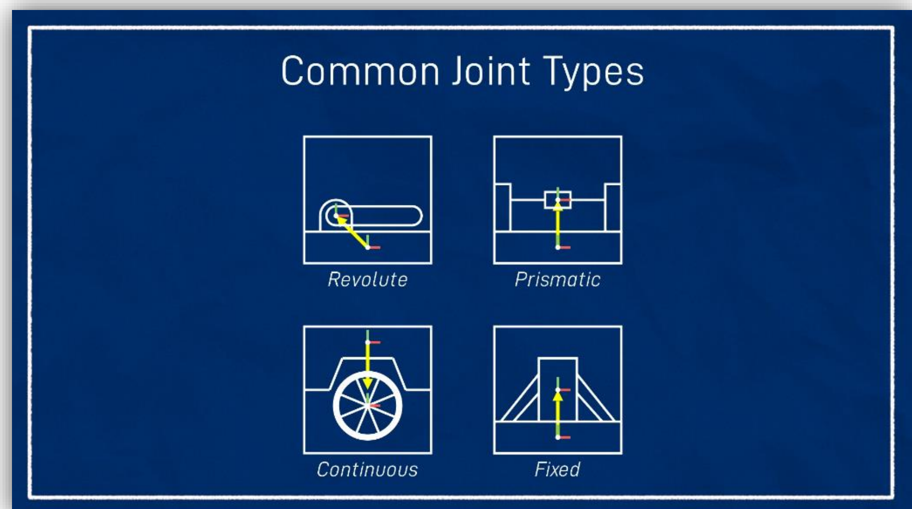

Ilustración 38: Descripción gráfica de las uniones

Para los enlaces también existen ciertas etiquetas que se deben poner, en este caso estas etiquetas describirán las características físicas del robot.

Etiquetas de los enlaces

- Visual: Esta etiqueta será la que especifique lo que veamos en Rviz o Gazebo.
  - Geometry: Tienes la opción de poner *box, cylinder o sphere* indicando cual va a ser el tamaño.
  - Origin: El origen de la geometría
  - Material: El color que va a tener
- Collision: Se utiliza para los cálculos de las colisiones
- Inertial: Se utiliza para calcular como va a responder el enlace a una fuerza ejercida sobre él.

```
<link name="arm_link">
    <visual>
        <geometry>
        <origin>
        <material>
    </visual>
    <collision>
        <geometry>
        <origin>
    </collision>
    <inertial>
        <mass>
        <origin>
        <inertia>
    </inertial>
</link>
```

Ilustración 39: Ejemplo de la descripción de un enlace

Una herramienta muy útil para crear tu robot con URDF es un visor online que lo que hace es ir representado el modelo del robot a medida que lo vas programando.

3.5.1 Xacro

Xacro es una versión mejorada de URDF con el que puedes crear variables dentro del mismo código o importar otros archivos Xacro o URDF para no tener que repetir el mismo código

3.5.2 Creación de un robot

Se ha realizado el programa "*brazo_robot.urdf*" (anexo 1) para ejemplificar todo lo explicado anteriormente y también para saber como darle uso.

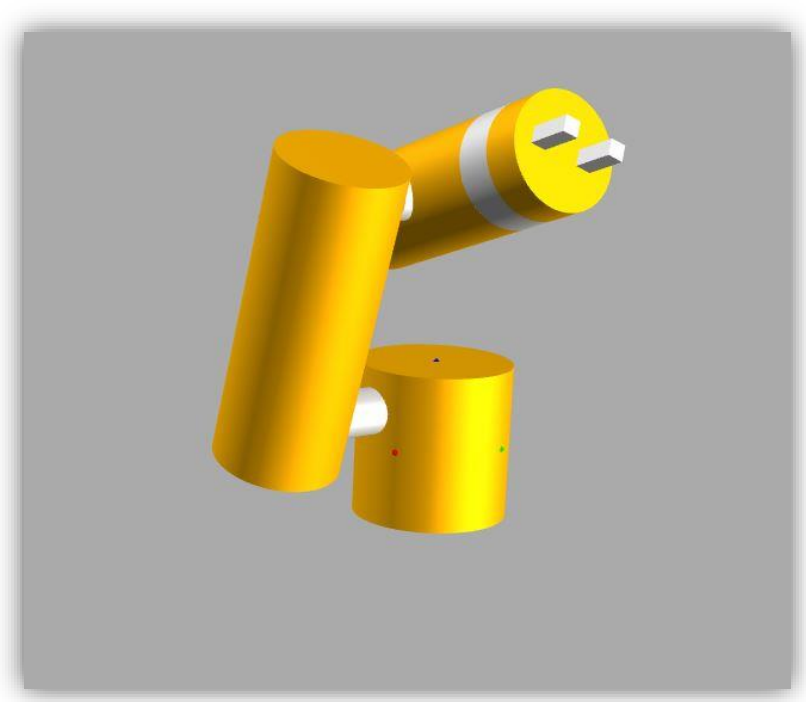

Ilustración 40: Representación Online de mi robot

Para poder visualizar el robot atreves de Gazebo o Rviz, o para poder programarlo y utilizarlo en un entorno de ROS hay que utilizar un asistente que proporciona el paquete de Moveit.

Para usar este asistente hay que usar este comando:

roslaunch moveit_setup_assistant setup_assistant.launch

Al ejecutar este comando aparecerá la siguiente pantalla:

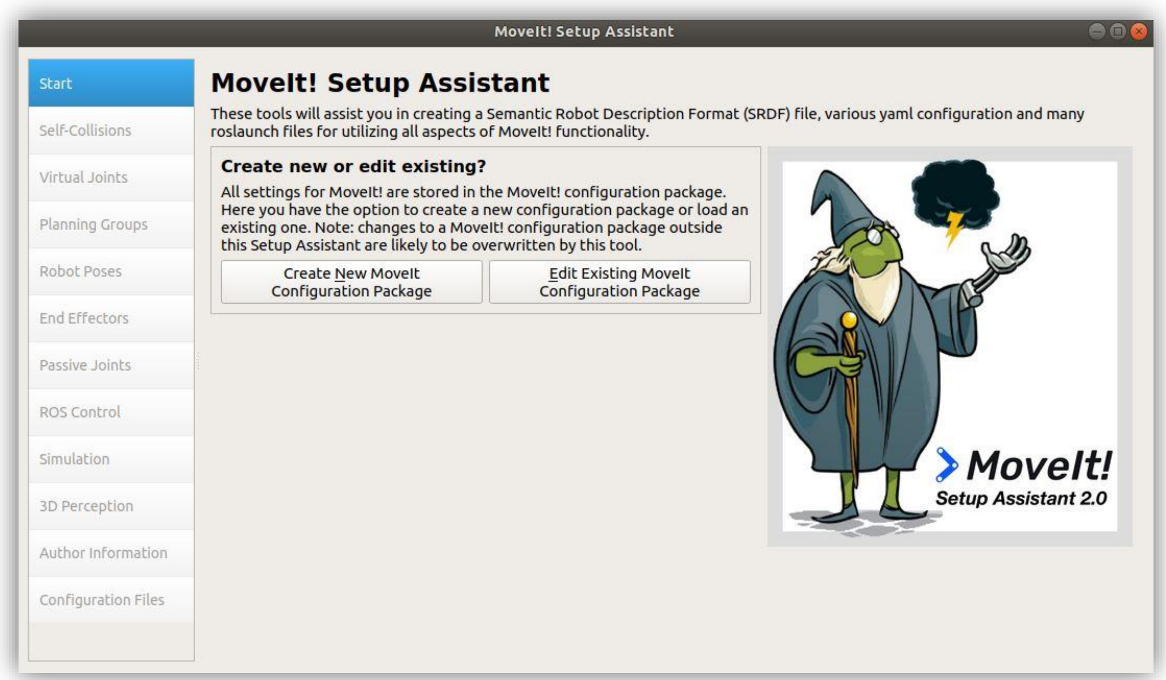

Ilustración 41: Interfaz del asistente de Moveit

Para poder cargar el archivo URDF en el asistente de Moveit primero hay que asegurarse de que este bien estructurado ya que si no es así el asistente nos dará un error. Atreves del comando **check_urdf brazo_robot.xml** se podrá saber si el archivo se podrá cargar o no.

```
ubuntu@ubuntu1804:~$ check_urdf brazo_robot.xml
robot name is: BRAZO_ROBOT
---------- Successfully Parsed XML ---------------
root Link: WORLD has 1 child(ren)
    child(1):  BASE_E1
        child(1):  U_BASE_E2
            child(1):  E2
                child(1):  U_E2_E3
                    child(1):  E3
                        child(1):  U_E3_E4
                            child(1):  E4
                                child(1):  G1
                                    child(1):  G2
                                child(2):  G2
```

Ilustración 42: Respuesta de check_urdf

Además si el archivo URDF está bien estructurado se podrán generar un pdf que muestre de forma visual las conexiones realizadas. Para ello se tendrá que ejecutar el comando **urdf_to_graphiz brazo_robot.xml.**

Al introducir el fichero URDF y nos permitirá ajustar algunas características, como crear un conjunto de joints o añadir un controlador determinado.

Antes de generar todos los paquetes elegidos debemos preparar nuestro entorno de trabajo. Al igual que se hizo con el robot Niryo Ned, he creado una carpeta con el nombre catkin_ws_brazorobot, dentro de ella he creado la carpeta “src” y otra carpeta dentro de “src” con el nombre “brazo_robot_package”.

Finalmente se da a generar paquetes en la carpeta “brazo_robot_package” y ejecutamos el comando ***catkin_make*** para preparar el espacio de trabajo de ROS.

Para lanzar el robot y poder visualizarlo con Rviz se ejecuta el comando ***roslaunch brazo_robot_package demo.launch.***

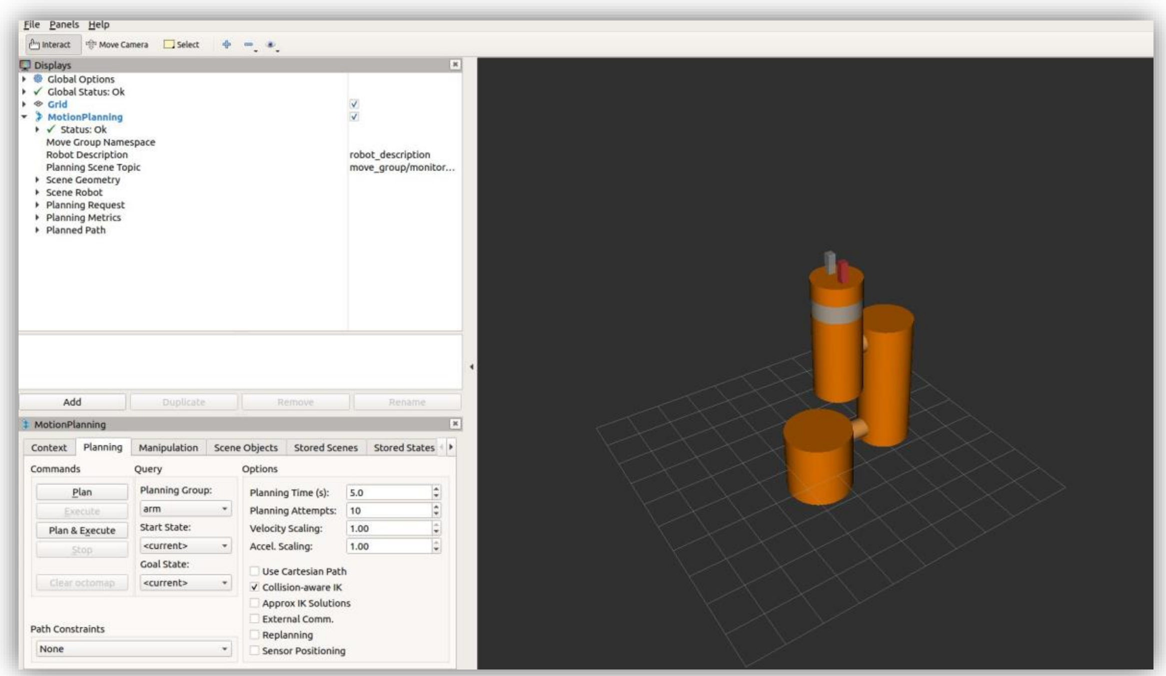

Ilustración 43: Representación en Moveit de mi robot

4. Ejemplos de programación en distintos niveles

En este apartado mostraré como aplicar todos los conceptos que he explicado en los anteriores apartados. Para ello realizaré unos programas que, de forma progresiva, irán completándose y haciéndose más complejos.

Para realizar los programas utilizaremos Python ya que es un lenguaje de programación que está en auge y es relativamente sencillo de entender. Por esta razón utilizaremos ROSpy. ROSpy es una paquete de Python que permite usar ROS con Python y también utilizaremos Moveit.

4.1 Programación con Pyniryo

Ilustración 44: Logotipo de PyNiryo

Otra de las formas que hay para controlar el robot es mediante el paquete de Python PyNiryo. Con este paquete la complejidad del programa residirá en lo que el usuario quiera programar puesto que PyNiryo te proporciona funciones fáciles de utilizar y que de una forma muy intuitiva se pueden entender.

Al ser más fácil realizar la programación el usuario tiene la posibilidad de centrar sus esfuerzos en otras metas.

Para mostrar el alcance que puede tener PyNiryo se han programado 2 ejemplos de control manual del Niryo Ned.

4.1.1 Control básico del NiryoNed

En este apartado se van a indicar las principales funciones que proporciona Pyniryo mediante las que se va a conseguir tanto mover el robot como obtener información de él.

```
from pyniryo import *

robot_ip_address= "127.0.0.1"
niryo_robot = NiryoRobot(robot_ip_address)
niryo_robot.calibrate(CalibrateMode.AUTO)

niryo_robot.move_joints([0.2, 0.1, 0.3, 0.0, 0.5, 0.0]) #(j1,j2,j3,j4,j5,j6)
niryo_robot.get_joints()

niryo_robot.move_pose([0.2, 0.1, 0.3, 0.0, 0.5, 0.0])#(x,y,z,roll,pitch,yaw)
niryo_robot.get_pose()

niryo_robot.open_gripper()
niryo_robot.close_gripper()
```

Ilustración 45: Código de movimientos del robot mediante Pyniryo

También existen funciones relacionadas con la visión artificial con las que podrás interactuar con la cámara que tiene el robot. Con el siguiente programa se consigue coger un objeto mediante la visión artificial.

```
def vision_pick_n_place_1(niyro_robot):
    intentos_exito = 0
    while intentos_exito < 5:
        # Se mueve a la posicion de observación
        niyro_robot.move_pose(*observation_pose.to_list())
        # Buscando un objeto que coger
        ret = niyro_robot.get_target_pose_from_cam(workspace_name,
                                                   height_offset=0.0,
                                                   shape=ObjectShape.ANY,
                                                   color=ObjectColor.ANY)
        obj_found, obj_pose, shape, color = ret
        if not obj_found:
            intentos_exito += 1
            continue

        niyro_robot.pick_from_pose(obj_pose)


        niyro_robot.place_from_pose(place_pose)
        break
```

Ilustración 46: Pick and place mediante visión artificial

4.1.2 Control manual mediante el mando de la Xbox360

Ilustración 47: Mando Xbox360

Al utilizar los controles básicos que nos proporciona Pyniryo hace que se pueda añadir al proyecto nuevas características que se complementan con las que ya hay.

La conexión que hay entre el mando y el equipo es atreves de un cable USB y la prueba se ha realizado en una máquina virtual con Ubuntu 18.04.

Para poder leer los datos que envía el mando con un programa de Python hay que ejecutar los siguientes comandos.

sudo apt-get install xboxdrv

sudo xboxdrv --detach-kernel-driver

git clone https://github.com/martinohanlon/XboxController

El primer comando lo que hace es instalar un programa cuya función es recopilar toda la información que el mando está transmitiendo.

El segundo comando se tiene que ejecutar antes de ejecutar el archivo Python puesto que es el encargado de iniciar el programa mencionado anteriormente.

A continuación, se va a mostrar la funcionalidad que se ha asignado a cada botón mediante el programa mostrado en el anexo 2:

Botón	Funcionalidad
X	Guardar posición actual
Y	Acceder al menú de posiciones
A	Abrir/Cerrar pinza
Select	Posición HOME
⬆	Cambiar eje a controlar
⬇	Cambiar eje a controlar
RB	Aumento de giro en cada movimiento
LB	Disminución de giro en cada movimiento
RT	Movimiento positivo del eje
LT	Movimiento negativo del eje

4.2 NiryoStudio

NiryoStudio es un software que permite controlar el robot de una forma sencilla y visual. A diferencia de los otros métodos de control del Niryo este programa facilita mediante una interfaz gráfica todas las opciones que tiene para ejecutar el robot.

Una vez conectado al robot introduciendo en el programa la dirección IP donde está alojado el Niryo Ned en nuestra pantalla aparecerá toda la información sobre el robot.

Ilustración 48: Datos proporcionados por NiryoStudio

En la parte superior derecha aparecerá una simulación en tiempo real del robot la cual nos servirá como guía para saber el estado en el que se encuentra en todo momento.

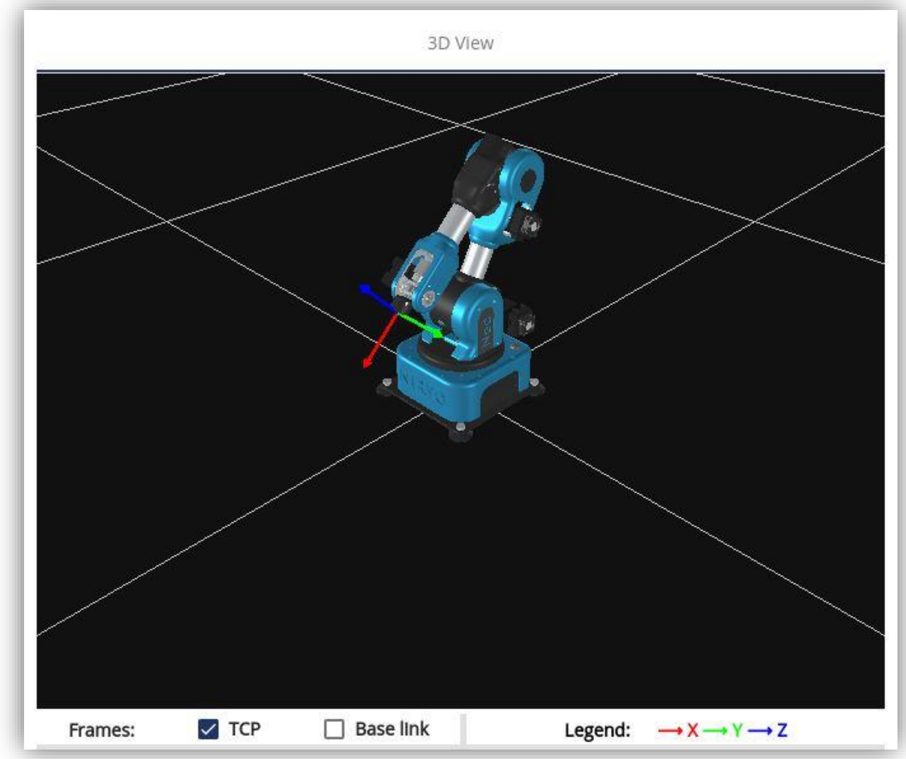

Ilustración 49: Visión en 3D en tiempo real

En la parte superior izquierda de la pantalla se dispone de hasta 4 opciones de control:

1. Mediante cursores que permiten modificar el valor de los 6 joints del robot y cuando se tenga la configuración elegida se aplica el movimiento

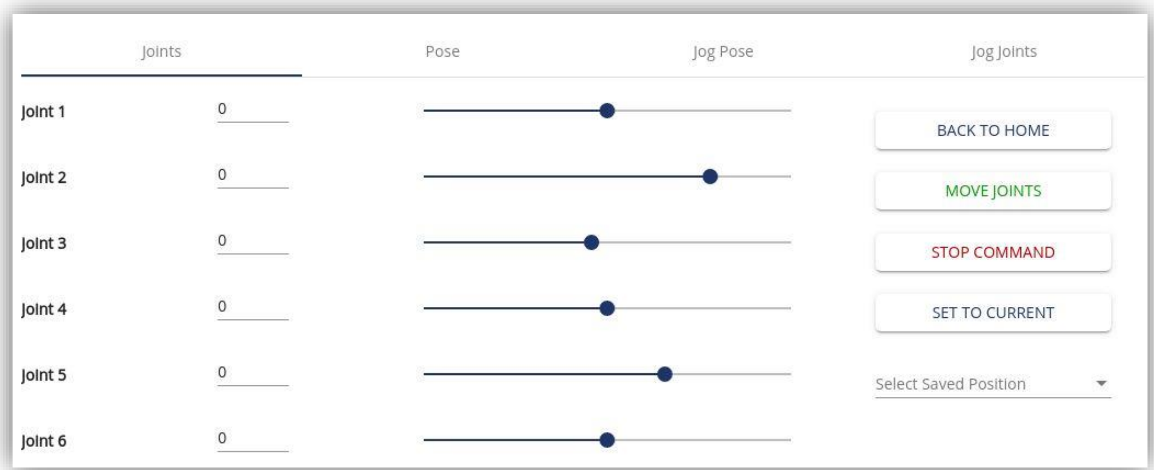

Ilustración 50: Cursores para editar el valor de cada Joint

2. Indicando la posición del TCP

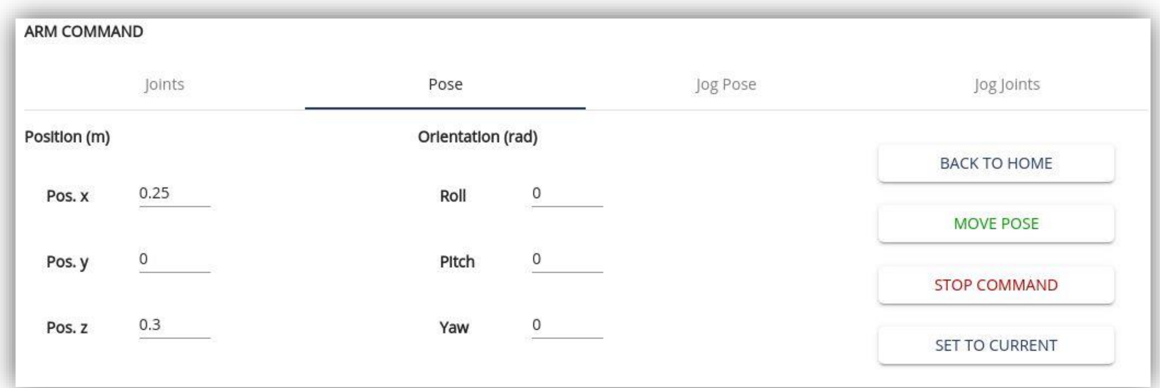

Ilustración 51: Posición del TCP

3. Atreves de flechas que irán moviendo en tiempo real la posición del TCP

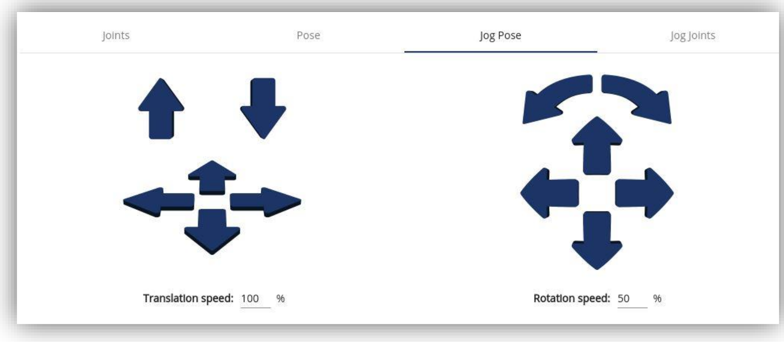

Ilustración 52: Movimiento del TCP mediante flechas

4. Esta forma es similar a la primera con la diferencia de que el movimiento del eje se hace en tiempo real.

Ilustración 53: Edición en tiempo real de cada Joint

Por último en la parte inferior izquierda se tiene la opción de configurar las entradas y salidas digitales.

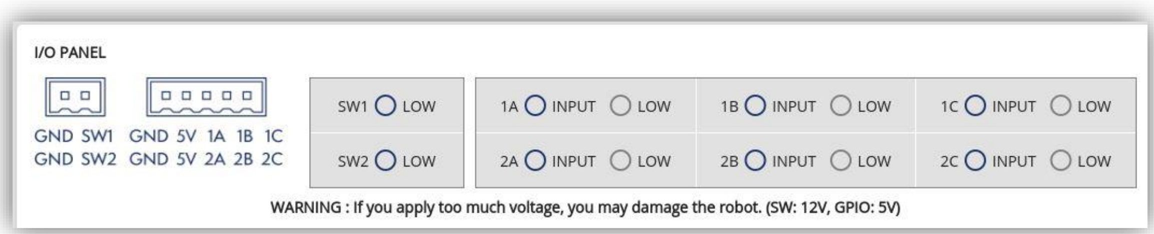

Ilustración 54: Interfaz de configuración de las entradas y salidas digitales

NiryoStudio también ofrece la opción de programar el robot mediante bloques. Este estilo de programas está sobretodo destinado para gente principiante que todavía no tiene nociones suficientes de programación. Aun así se puede conseguir un control muy similar y de una forma muy intuitiva.

Estos son los bloques más importantes:

- Bloques lógicos
- Bloques de movimientos
- Bloques de visión
- Bloques de interacción con la herramienta
- Bloques de variables

4.3 Comunicación entre 2 nodos atreves de un topic

En este primer ejemplo es lo más básico posible, consiste en crear 2 nodos y que se comuniquen entre ellos. Uno de los nodos será el nodo_emisor y el otro el nodo_receptor. Para que ambos se puedan comunicar, el primero estará suscrito al topic con el nombre canal_1 y el segundo publicará el mensaje en ese mismo topic.

Antes de crear el programa debemos crear el paquete que contendrá nuestro archivos de python. Para ello la forma más indicada es utilizando el comando ***catkin_create_pkg <nombre del paquete> std_msgs rospy roscpp*** .

Los complementos que se han puesto a continuación del nombre del paquete son dependencias necesarias para que haya un buen funcionamiento del entorno de ROS. Esta dependecias pueden variar en función de los requerimientos o funciones que vaya a realizar el paquete.

Una vez creado se añadira en la ubicación donde te encuentres una carpeta con el nombre del paquete. Dentro de esta carpeta habrá otras dos carpetas, "include" y "src" y dos archivos necesarios para que se pueda ejecutar el comando catkin_make.

Dentro de la carpeta src será donde se guardarán los programas de python

Una vez creados los programas (anexos 3 y 4) se deberá ejecutar el comando ***catkin_make*** para que guarde todo dentro del entorno de ROS.

```
def main():
    rospy.init_node("nodo_emisor",anonymous=True)
    pub=rospy.Publisher("canal_1",String,queue_size=10)
    rate=rospy.Rate(10)
    while not rospy.is_shutdown():
        msg="MENSAJE"
        pub.publish(msg)
        rate.sleep()




if __name__ == '__main__':
    main()
```

Ilustración 55: Programa nodo emisor

```
def listen_msg(data):
    rospy.loginfo(data)

def main():
    rospy.init_node("nodo_receptor",anonymous=True)
    rospy.Subscriber("canal_1",String,listen_msg)
    rospy.spin() #Evita salir del programa hasta que se cierre el nodo


if __name__ == '__main__':
    main()
```

Ilustración 56: Programa nodo receptor

Para ejecutar los programas la mejor opción es utilizar el comando ***rosrun***

<nombre del paquete> <nombre del archivo de python>

Una vez ejecutado ambos programas podemos ver que al ejecutar el comando ***rosnode list*** aparecen los nodos que hemos creado.

```
ubuntu@ubuntu1804:~$ rosnode list
/nodo_emisor_4663_1669305631356
/nodo_receptor_5185_1669305793036
/rosout
```

Ilustración 57: Listado de nodos activos

Por último, la comunicación es exitosa debido a que el nodo_receptor está recibiendo el mensaje.

```
ubuntu@ubuntu1804:~$ rosrun niryo_datos sub.py
[INFO] [1669305642.393921]: data: "MENSAJE"
[INFO] [1669305642.493179]: data: "MENSAJE"
[INFO] [1669305642.593621]: data: "MENSAJE"
[INFO] [1669305642.693947]: data: "MENSAJE"
[INFO] [1669305642.793836]: data: "MENSAJE"
[INFO] [1669305642.893144]: data: "MENSAJE"
[INFO] [1669305642.993350]: data: "MENSAJE"
[INFO] [1669305643.093156]: data: "MENSAJE"
[INFO] [1669305643.193068]: data: "MENSAJE"
[INFO] [1669305643.293862]: data: "MENSAJE"
```

Ilustración 58: Ejecución del nodo receptor

4.4 Acceso a datos de un nodo.

En este ejemplo se va a tratar de recopilar datos del robot. Para ello se va a crear un nodo que va a estar suscrito a uno de los topics que se generan cuando el robot Niryo Ned se está simulando.

Para simular el robot tenemos la opción de hacerlo por Gazebo o por Rviz, el simularlo de formas distintas hará que se generen nodos y topics distintos ya que cada simulador crea lo que necesita para funcionar. Pero también hay nodos que van a estar en ambas simulaciones que son los que genera el Niryo Ned atreves de los archivos que se descargaron.

Para este caso utilizaremos Gazebo y al ejecutarlo obtenemos la siguiente lista de nodos.

```
ubuntu@ubuntu1804:~$ rosnode list
/controller_spawner
/gazebo
/gazebo_gui
/move_group
/move_group_commander_wrappers_1669462941599476717
/moveit_python_wrappers_1669462941642068193
/niryo_robot_arm_commander
/niryo_robot_hardware_interface
/niryo_robot_modbus
/niryo_robot_poses_handlers
/niryo_robot_programs_manager
/niryo_robot_rpi
/niryo_robot_status
/niryo_robot_system_api_client
/niryo_robot_tools_commander
/niryo_robot_user_interface
/niryo_robot_vision
/robot_state_publisher
/rosapi
/rosbridge_websocket
/rosout
/tf2_web_republisher
```

Ilustración 59: Listado de nodos activos con Gazebo

Como se puede apreciar hay dos nodos de Gazebo, también está el nodo move_group y los propios nodos del robot.

A partir de este momento tenemos que seguir un orden para poder encontrar lo que queremos.

1. Tener claro los datos que estamos buscando
2. Buscar en la documentación del robot que nodo o topic nos puede proporcionar los datos que estamos buscando o ejecutar el comando ***rostopic list*** y buscar un topic con el nombre que se pueda parecer a lo que queremos.
3. Comprobar con el comando ***rostopic echo*** si los mensajes que se publican se ajustan lo que queremos

Para poner en práctica este procedimiento voy a ejemplificarlo con la obtención del estado de los joints.

Como ya sabemos lo que vamos a buscar ahora ejecutaré el comando ***rostopic list*** para ver si hay algún topic que se ajuste a lo que busco.

Al realizar la búsqueda en la lista obtenida encontré el topic llamado /joint_states.

Ahora ejecuto el comando ***rostopic echo /joint_states*** y compruebo el resultado.

```
name: [joint_1, joint_2, joint_3, joint_4, joint_5, joint_6, joint_base_to_mors_
1, joint_base_to_mors_2]
position: [0.01715443767025881‌5, 0.13785982136512942, 0.03929289858496254, -0.02
0729773303684418, -0.010006562706194266, 0.019951489762456198, 0.009768754321394
058, 0.009884479253093957]
velocity: [2.6591950358854147e-05, -3.25501211393935e-05, 9.08582145685139‌2e-06,
 -3.607066804374804e-05, 3.837151688799987e-06, 2.8042253816583096e-06, 3.907954
814448106e-06, -3.9079786881536106e-06]
effort: []
---
```

Ilustración 60: Mensaje publicado en el topic /joint_states

En la ilustración 47 se puede ver que no solo obtenemos los joints sino que también obtenemos velocidades.

Tras haber comprobado esta parte ya podemos proceder a programarlo en Python. Siguiendo el primer ejemplo necesitamos crear un nodo que se suscriba al topic /joint_states para que el mismo muestre por la terminal los datos obtenidos.

```python
import rospy
from sensor_msgs.msg import JointState

def listen_msg(data):
    rospy.loginfo(data)

def main():
    rospy.init_node("nodo_receptor",anonymous=True)
    rospy.Subscriber("joint_states",JointState,listen_msg)
    rospy.spin() #Evita salir del programa hasta que se cierre el nodo


if __name__ == '__main__':
    main()
```

Ilustración 61: Programa para suscribirse al nodo joints_states

Se puede apreciar una leve diferencia entre esta ilustración y la ilustración 43 y es que el tipo de información que se va a intercambiar es distinta.

Para saber que tipo de mensaje se envía se ha ejecutado el comando ***rostopic info /joint_states,*** al hacerlo nos muestra que el tipo de mensaje que se intercambia es "sensor_msgs/JointState". Como este tipo de variable no es propia de Python hay que importarle.

4.5 Creación y uso de un servicio con programación mediante paquetes de Moveit y Rospy

En este ejemplo se van a crear 2 nodos, uno va a tener el nombre "niryo_menu" y el otro "niryo_mov", el primero se va a encargar de mostrar por pantalla todas las opciones que vamos a tener para interaccionar con el robot y segundo tiene la finalidad de ejecutar los datos tal y como dice el primer nodo.

Para la comunicación entre ambos nodos he creado un servicio cuya estructura está compuesta por el mensaje que se envía y por lo que se recibe de respuesta. He decidido que la comunicación entre ambos nodos sea mediante un servicio ya que el intercambio de información va a ser aperiódico.

Para crear el servicio hay que seguir un orden, ya que aunque no es complejo es fácil que a la primera no funcione.

1. Para nuestro caso debemos generar dos paquetes, los cuales les vamos a generar mediante los siguiente comandos

catkin_create_pkg niryo_menu rospy std_msgs message_generation roscpp

catkin_create_pkg niryo_mov rospy std_msgs message_generation roscpp

En este caso al crear los paquetes hemos añadido una dependencia más que permitirá que podamos usar los servicios que se guarden en cada paquete.

2. Dentro de cada paquete debemos crear una carpeta con el nombre "srv" la cual tendrá el archivo .srv que es donde se indica la estructura del servicio.

También hay que crear un archivo de Python con el nombre setup.py con el siguiente contenido:

```
from setuptools import setup
from catkin_pkg.python_setup import generate_distutils_setup

d = generate_distutils_setup(
    packages=['niryo_menu'],
    package_dir={'': 'src'}
)

setup(**d)
```

Ilustración 63: Setup.py del nodo menú

```
from setuptools import setup
from catkin_pkg.python_setup import generate_distutils_setup

d = generate_distutils_setup(
    packages=['niryo_mov'],
    # scripts=['bin/myscript'],
    package_dir={'': 'src'}
)

setup(**d)
```

Ilustración 62: Setup.py del nodo mov

3. Si los paquetes se han generado bien en cada uno habrá un archivo con el nombre CMakeLists.txt. En ellos se va a tener que realizar una modificación que consistirá en añadir las siguientes líneas:

```
add_service_files(
        DIRECTORY srv
        FILES
                msg_prot.srv
)


catkin_python_setup()
```

Con esto lo que se consigue es que cuando se realice el catkin_make el entorno de ROS sepa donde se encuentra el archivo msg_prot.srv.

4. En la carpeta srv creamos un archivo con el mismo nombre que hemos puesto en el CMakeLists.txt, en este caso se ha puesto msg_prot.srv
   Este archivo contendrá las siguientes líneas.

   string msg
   ---
   bool conf
   string resp

   Mediante esta estructura lo que se está indicando es que la parte superior, que se corresponde a la petición que realiza el nodo niryo_menu, es un string con el nombre msg. La parte inferior es la respuesta que da el nodo niryo_mov que esta formada por una variable booleana que indicará si la petición se ha podido realizar y por un string que se utilizará para dar información complementaria a lo que haya sucedido con la petición.

5. La última modificación que hay que realizar es en el archivo con el nombre package.xml el cual también se ha tenido que generar en el momento en que se creó el paquete.
   Hay que añadirle la siguiente línea:

   <exec_depend>message_runtime</exec_depend>

6. Una vez realizado estos cambios debemos ejecutar el comando ***catkin_make***

4.5.1 Nodo "niryo_mov"

Para utilizar el este servicio con Rospy debemos añadirlo de la siguiente forma:

```
rospy.init_node("niryo_mov",anonymous=True)
rospy.Service("/comunication_service",msg_prot,listen_comunication)
rospy.Subscriber("/niryo_robot/robot_state",RobotState,listen_RS)
rospy.Subscriber("/joint_states",JointState,listen_status)
rospy.Subscriber("/niryo_robot/collision_detected",Bool,listen_colision)
```

Ilustración 64: Configuración del nodo mov

De esta manera ya tenemos garantizado que el niryo_mov va a poder utilizar el servicio bajo el nombre de "comunication_service".

La función asignada para gestionar el servicio va a recibir el mensaje y en función de lo que se pueda hacer dará una respuesta:

```
def listen_comunication(data):
    global msg_srv
    global step
    msg_srv=data.msg
    conf,resp=proceso(msg_srv)
    return msg_protResponse(conf,resp)
```

Ilustración 65: Función encargada de gestionar el servicio

La función proceso es la que se encarga de dar funcionalidad al mensaje que ha llegado del nodo "niryo_menu" y cuando termina de hacerlo devuelve si se ha podido realizar y una respuesta.

Para que se puedan ejecutar los movimientos con el robot se utilizarán librerías de Moveit para Python. Estas librerías nos permitirán programar el robot de una manera muy simple, parecida al software Rapid de ABB o softwares similares.

También se va a utilizar una librería que viene incluida con todos los paquetes descargados del Niryo Ned. Esta librería se llama NiryoRosWrapper y nos permitirá poder interaccionar con la cámara del robot. Esta librería también nos permite mover el robot de una forma similar que el Moveit pero no se ha programado de esta forma ya que es una librería exclusiva del Niryo Ned y el programa no sería extrapolable a otros robots.

Finalmente todas las funciones y objetos que vamos a importar son los siguientes:

```
import moveit_commander
from lee_file import prepara_fichero
import rospy
from niryo_mov.srv import msg_prot,msg_protResponse
from niryo_robot_msgs.msg import RobotState
from sensor_msgs.msg import JointState
from niryo_robot_python_ros_wrapper import *
from std_msgs.msg import Bool
from geometry_msgs.msg import PoseStamped
```

Ilustración 66: Importaciones necesarias

Moveit_commander es la librería que nos va a permitir interaccionar con el robot mediante movetit, prepara_fichero es una función cuya finalidad es leer de un .txt unas posiciones guardadas del robot. El resto son objetos necesarios para poder intercambiar datos mediante publicaciones en topics o con servicios.

Del programa voy a comentar lo que considero más importante y sobre todo lo que sería más útil a la hora de crear un programa nuevo.

Para utilizar Moveit debemos indicar de la siguiente manera la forma en que el nodo move_group debe buscar tanto el brazo como la pinza dentro de los paquetes del Niryo Ned.

Para conocer esta información debemos acudir al paquete con nombre “niryo_moveit_config_standalone” (en este caso se accede en esta carpeta, pero he podido ver que para otros robots la información que buscamos se encuentra normalmente en la carpeta description), dentro de ella accedemos a “config”, “ned” y en esta carpeta tenemos el archivo .srdf que contiene los datos que se están buscando.

```
    <!--SUBGROUPS: Groups can also be formed by referencing to already
defined group names-->
    <group name="arm">
        <chain base_link="base_link" tip_link="tool_link" />
    </group>
    <group name="tool">
        <link name="tool_link" />
```

Ilustración 67: Nombres de los subgrupos del NiryoNed

Los dos subsgrupos con nombre “arm” y “tool” van a ser los nombres que indiquemos a Moveit para que pueda acceder a los datos estructurales del Niryo Ned.

```
robot = moveit_commander.MoveGroupCommander("arm")
gripper=moveit_commander.MoveGroupCommander("tool")
escena=moveit_commander.PlanningSceneInterface()
```

Ilustración 68: Inicialización de los objetos de Python

El objeto escena lo voy a utilizar para describir como es el entorno donde se encuentra el robot. De esta manera cuando ejecute movimientos de forma automática evitará chocarse con ellos.

Para crear este entorno he creado una función que crea el entorno y posteriormente va actualizándose en función de los cambios que hayan ocurrido.

```
def crea_colision():
    escena.clear()
    escena.remove_attached_object(gripper.get_planning_frame())
    cubo=PoseStamped()
    cubo.header.frame_id=gripper.get_planning_frame()
    for morfo in forma:
        for c in color:
            obj=(niryo_robot.detect_object(ws,morfo,c))
            if obj[0]==True:
                obj_pose_world=niryo_robot.get_target_pose_from_rel(ws,0.05,obj[1].x,obj[1].y,obj[1].yaw)
                cubo.pose.position.x=obj_pose_world.position.x
                cubo.pose.position.y=obj_pose_world.position.y
                cubo.pose.position.z=obj_pose_world.position.z
                cubo.pose.orientation.x=cubo.pose.orientation.y=cubo.pose.orientation.z=0
                cubo.pose.orientation.w=1
                cubo_name=obj[2]+obj[3]
                escena.attach_box(eef_link,cubo_name,cubo,SQUARE_size)
    cilindro=PoseStamped()
    cilindro.header.frame_id=robot.get_planning_frame()
    cilindro.pose.position.x=0.25
    cilindro.pose.position.y=-0.014
    cilindro.pose.position.z=0.05
    cilindro.pose.orientation.x=cilindro.pose.orientation.y=cilindro.pose.orientation.z=0
    cilindro.pose.orientation.w=1
    cilindro_name='cilindro'
    escena.add_cylinder(cilindro_name,cilindro,0.15,0.02)
```

Ilustración 69: Función encargada de crear el entorno de simulación

Esta función añade al entorno todos los cubos que la cámara haya encontrado y un cilindro cuya posición no va a variar nunca. De esta parte del programa quiero destacar la diferencia entre la attach y add del objeto escena.

Con attach lo que se está diciendo es que son objetos que en un determinado momento pueden ser cogidos por la pinza por lo que se permite la colisión entre la pinza y el cubo. Mientras que si hubiéramos añadido los cubos de la misma forma que el cilindro el robot detectaría colisión.

4.5.2 Nodo "niryo_menu"

El programa para este nodo es mucho más sencillo ya que solo consta de la parte que se encarga de mostrar por la terminal las opciones del menú y la que gestiona la comunicación.

```
rospy.wait_for_service("comunication_service")
srv=rospy.ServiceProxy("/comunication_service",msg_prot)
```

Ilustración 70: Creación del servicio en el nodo menú

Mediante estas dos líneas lo que se consigue es que el nodo no comience a funcionar hasta que el nodo "niryo_mov" haya creado el servicio, el objeto srv va a ser el que realice la comunicación entre nodos.

Para que este nodo se comunique con el otro hay que hacerlo de la siguiente forma.

resp=srv("mensaje")

Como el propio nombre dice lo que va entre paréntesis es el mensaje (string) que se va a enviar y resp será una tupla formada por un bool y un string, tal que y como se configuro en el archivo .srv.

4.5.3 Simulación con Gazebo

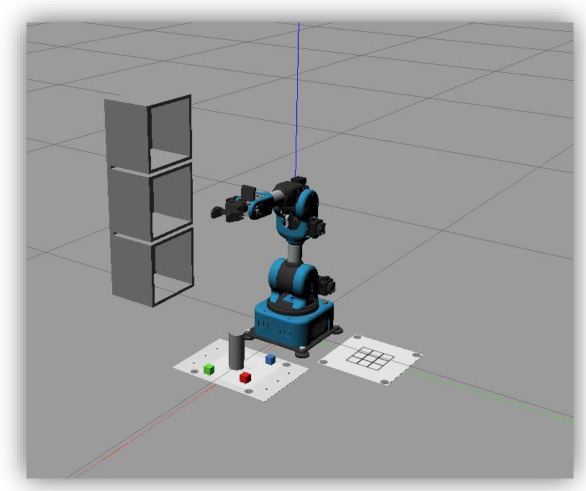

Ilustración 71: Entorno de simulación en Gazebo

Una vez lanzado el Gazebo con el entorno creado se ejecutan los nodos con los siguientes comandos en diferentes terminales:

rosrun niryo_mov moveit.py

rosrun niryo_menu menu.py

La terminal del nodo mov no es necesaria por lo que se puede minimizar y la pantalla del nodo menú es la siguiente:

```
ubuntu@ubuntu1804:~$ rosrun niryo_menu menu.py
------------MENU-------------
1) mov_unico
2) mov_sec
3) estado_act
4)Abrir/Cerrar gripper
5) Recoge_cubo
6) SALIR
```

Ilustración 72: Pantalla de nodo menú

Como se puede ver el menú ofrece 6 opciones:

1) mov_unico : Existe un fichero con unas posiciones guardadas y mediante esta opción podemos ubicar el robot en la que queramos

```
1
{'PosObv': [0.0, 0.3, -0.231, 0.0, -1.71, 0.0], 'HOME': [0.0, 0.512, -1.251, 0.0
, 0.0, 0.0], 'Pos4': [-1.5, -0.489, -0.749, 0.0, 0.952, 0.0], 'Pos5': [-1.5, -0.
811, 0.024, 0.0, 0.61, 0.0], 'Pos2': [-1.5, -0.194, -0.189, 0.0, 0.21, 0.0], 'Po
s3': [-1.5, -0.6, 0.4, 0.0, 0.463, 0.0], 'Pos0': [-1.5, -0.165, 0.341, 0.0, -0.0
, 0.0], 'Pos1': [-1.5, -0.705, 1.196, -0.232, -0.471, 0.0]}
HOME
movimiento realizado
-----------MENU-------------
```

Ilustración 73: Ejemplo de selección de la opción 1

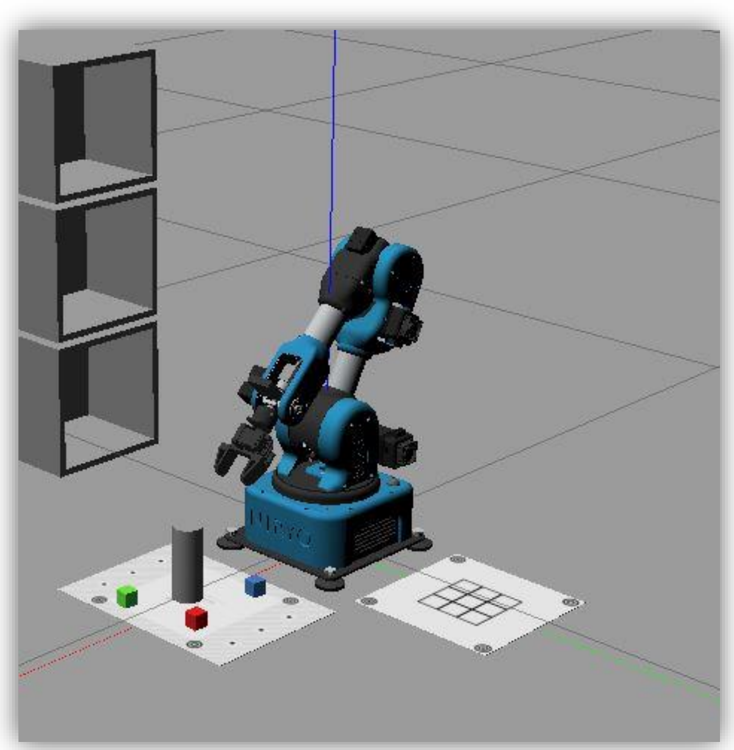

Ilustración 74: Robot en posición HOME

2) mov_sec: Mediante este comando se podrá ejecutar un movimiento secuencial entre las posiciones guardadas.

```
------------MENU-------------
1) mov_unico
2) mov_sec
3) estado_act
4)Abrir/Cerrar gripper
5) Recoge_cubo
6) SALIR
2

ELIJA UNA SECUENCIA SEPARANDO LAS POSICIONES POR COMAS
Pos0,Pos1,Pos0,PosObv
movimientos realizado
------------MENU-------------
```

Ilustración 75: Ejemplo de selección de la opción 2

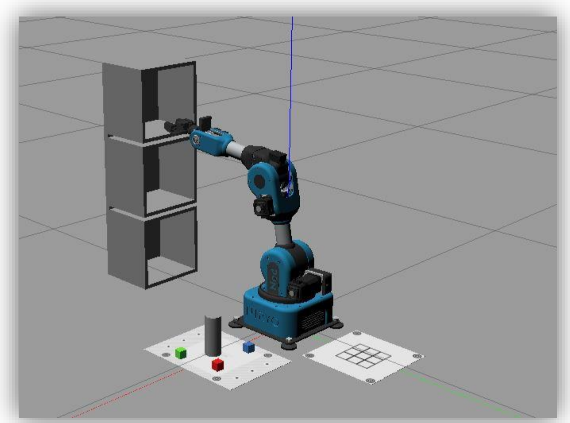

Ilustración 76: Robot en Pos0

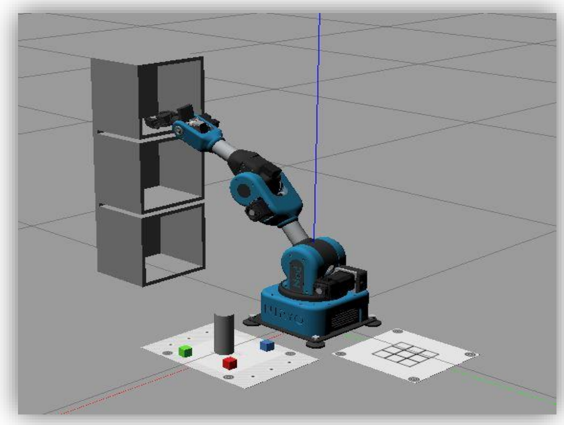

Ilustración 77: Robot en Pos1

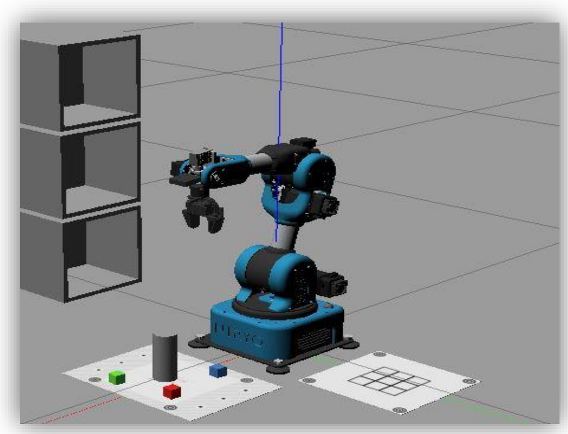

Ilustración 78: Robot en PosObv

3) estado_act: Con este comando se muestra por pantalla el valor de todos los joints (j6 y j7 son el grado de apertura de la pinza) y de la posición y orientación del robot

```
3
J0:(0.01004440121161565
J1:0.28714470716465357
J2:-0.24875607313080472
J3:-0.009760277514008209
J4:-1.7199211113331758
J5:0.010073255450475216
J6:0.009378004858235015
J7:0.009514210643688011)
roll: -3.04348799706
pitch: 1.45963507446
yaw: -3.04443201231
x: -0.00349579385985
y: 0.745108921982
z: -0.00416228263441
w: 0.666920646863
```

Ilustración 79: Ejemplo de selección de la opción 3

4) Abrir/Cerrar gripper: Al usar esta opción tendremos la posibilidad de abrir o cerrar el gripper.

```
------------MENU--------------
1) mov_unico
2) mov_sec
3) estado_act
4)Abrir/Cerrar gripper
5) Recoge_cubo
6) SALIR
4
1) Abrir gripper
 2) Cerrar gripper
```

Ilustración 80: Ejemplo de selección de la opción 4

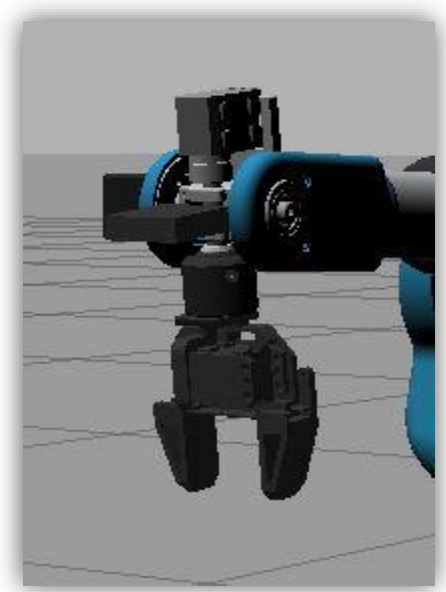

Ilustración 82: Gripper abierto

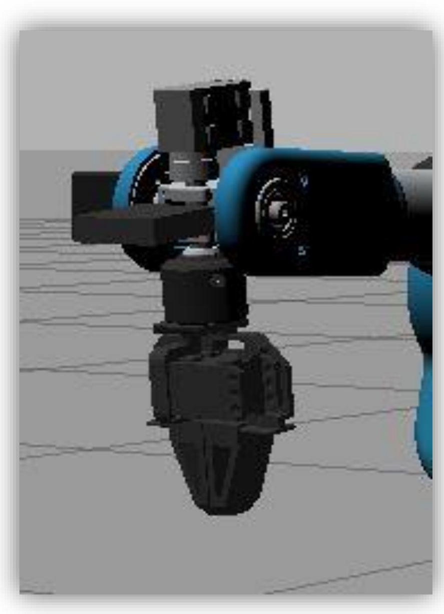

Ilustración 81: Gripper cerrado

5) Recoge cubo: Al elegir esta opción el robot comienza a buscar los bloques que hay en la plantilla e intentará localizarlos mediante la visión que le otorga la cámara, para ello deberá estar en la posición de observación (PosObv). Una vez localizados deberá cogerlos y transportarlos al armario sin chocarse con el cilindro.

```
robot.go(dic_pos.get("PosObv"))
crea_colision()
for morfo in forma:
    for c in color:
        obj=(niryo_robot.detect_object(ws,morfo,c))
        if obj[0]==True:
            pose_rel_obj_dist=niryo_robot.get_target_pose_from_rel(ws,0.08,obj[1].x,obj[1].y,obj[1].yaw)

            joint_rel_obj_dist=niryo_robot.inverse_kinematics(pose_rel_obj_dist.position.x,
            pose_rel_obj_dist.position.y,pose_rel_obj_dist.position.z,pose_rel_obj_dist.rpy.roll,
            pose_rel_obj_dist.rpy.pitch,pose_rel_obj_dist.rpy.yaw)
            robot.go(joint_rel_obj_dist)

            gripper.go(open_pose)
            pose_rel_obj=niryo_robot.get_target_pose_from_rel(ws,0,obj[1].x,obj[1].y,obj[1].yaw)
            joint_rel_obj=niryo_robot.inverse_kinematics(pose_rel_obj.position.x,pose_rel_obj.position.y,
            pose_rel_obj.position.z,pose_rel_obj.rpy.roll,pose_rel_obj.rpy.pitch,pose_rel_obj.rpy.yaw)

            robot.go(joint_rel_obj)
            gripper.go(close_pose)
            robot.go(dic_pos.get("Pos2"))
            robot.go(dic_pos.get("Pos3"))
            gripper.go(open_pose)
            robot.go(dic_pos.get("Pos2"))
            robot.go(dic_pos.get("PosObv"))
            escena.remove_attached_object(obj[2]+obj[3])
```

Ilustración 83: Código que se encarga de ejecutar la opción 5

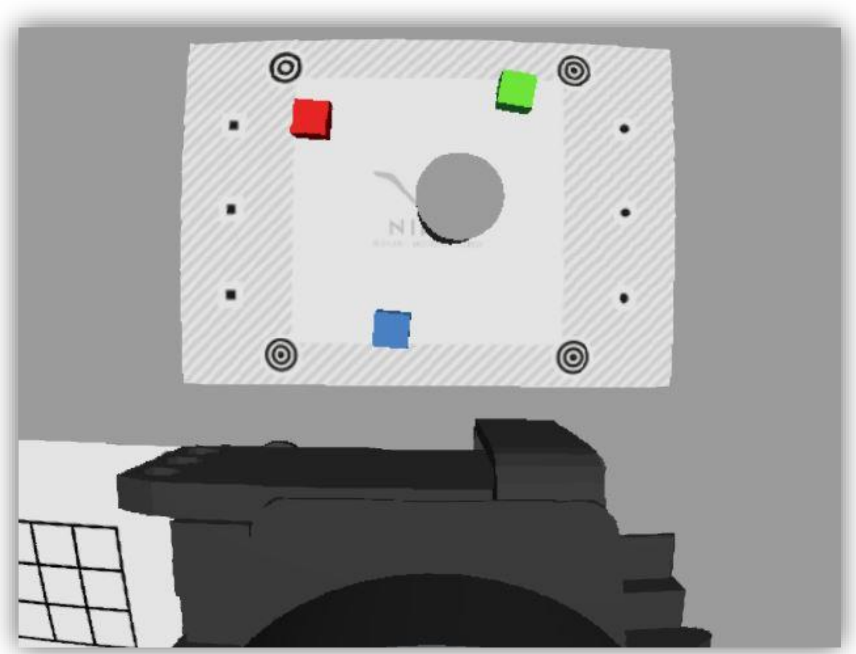

Ilustración 84: Visión de la cámara en PosObv

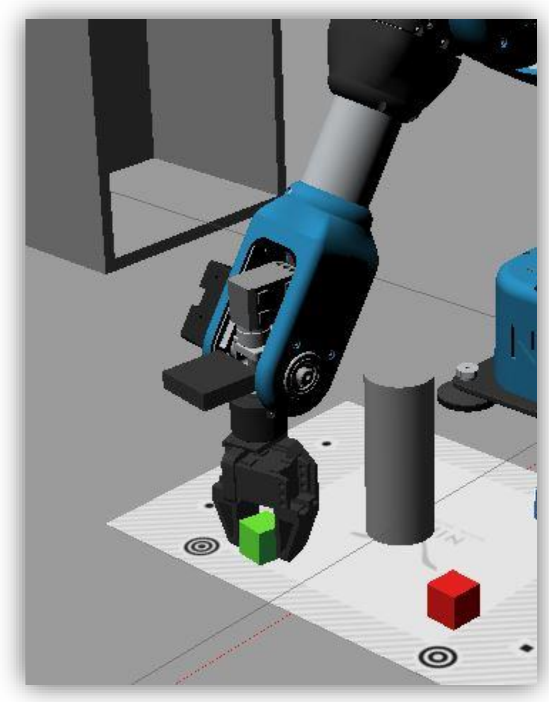

Ilustración 85: Captura del bloque verde

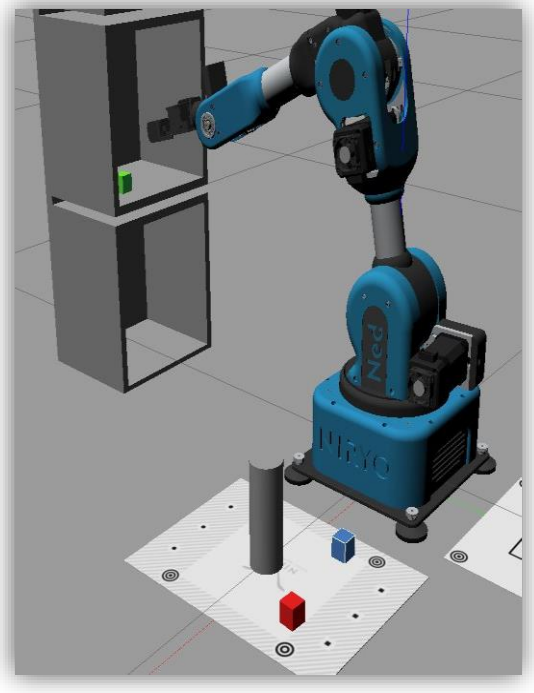

Ilustración 86: Bloque verde depositado en el armario

6) Salir: Esta opción se puede utilizar para cerrar de forma adecuada el nodo menú y posteriormente el nodo mov.

4.6 Aplicación del mismo código en 2 robots simulares.

Una de las ventajas que proporciona ROS es la capacidad de usar el mismo programa para un robot similar. En este caso se va a comparar el robot Niryo Ned con otro robot similar también de 6 ejes.

El programa que vamos a ejecutar es muy sencillo y consiste en mandar al robot a 3 posiciones de manera consecutiva.

```
import moveit_commander

robot = moveit_commander.MoveGroupCommander("arm") #Para mover el brazo
joint1=[-0.82,-0.44,-0.34,-0.64,-0.36,1.81]
joint2=[0.06,-0.45,-0.04,0,0.63,0]
joint3=[0,0.45,0,1.28,0.63,-1.1]


def main():
        robot.go(joint1)
        robot.go(joint2)
        robot.go(joint3)


if __name__ == '__main__':
    main()
```

Ilustración 87: Código utilizado para el NiryoNed

```
import moveit_commander

robot = moveit_commander.MoveGroupCommander("arm_group") #Para mover el brazo
joint1=[-0.82,-0.44,-0.34,-0.64,-0.36,1.81]
joint2=[0.06,-0.45,-0.04,0,0.63,0]
joint3=[0,0.45,0,1.28,0.63,-1.1]


def main():
        robot.go(joint1)
        robot.go(joint2)
        robot.go(joint3)



if __name__ == '__main__':
    main()
```

Ilustración 88: Código utilizado para el 2º robot colaborativo

La única diferencia entre ambos programas es el nombre que se le da al subgrupo, el cual es necesario que este bien puesto es lo que Moveit va a utilizar para buscar el URDF del robot.

5. Conclusión

Finalizado el TFG se ha llegado a un conjunto de conclusiones relacionadas con el uso de ROS y sus posibles aplicaciones.

Con el auge que se está viviendo hoy en día con la implementación de robots colaborativos en distintos aspectos, como puede ser en el campo industrial o en el de la medicina, ROS es una herramienta muy potente que ofrece un gran abanico de posibilidades para poder controlar de la mejor manera cualquier robot.

De todas estas posibilidades se debe destacar 2:

- El poder simular cualquier robot que disponga de los paquetes de ROS con Gazebo o Rviz, permite poder controlar el robot de una forma más óptima, debido a que antes de probarlo con el robot real se puede simular con todo tipo de detalles en el entorno de trabajo que se haya configurado.

- El que un programa que se ha destinado para un robot también se puede usar para otro robot similar lo consideró uno de los factores más importantes de por qué usar ROS. De esta forma se puede reciclar programas y en caso de no saber como hacer un determinado movimiento del robot hace que se pueda buscar ya que puede que alguien ya lo haya hecho.

Además mediante este TFG se va a poder realizar una extrapolación de los pasos y fundamentos básicos que se han explicado hacia otros robots que vaya adquiriendo la escuela con el fin de que la implementación y uso de ROS cada vez sea más fácil de entender.

6. Anexos

6.1 Anexo 1: Programa brazo_robot.urdf

```
1. <?xml version="1.0"?>
2.
3. <robot name="BRAZO_ROBOT" xmlns:xacro="http://www.ros.org/wiki/xa
   cro">
4.
5.     <link name="WORLD"/>
6.
7.     <link name="BASE_E1">
8.         <visual>
9.             <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0"/>
10.                <geometry>
11.                    <cylinder radius="1" length="2"/>
12.                </geometry>
13.                <material name="NARANJA">
14.                    <color rgba="1 0.5 0.0 1"/>
15.                </material>
16.            </visual>
17.            <inertial>
18.                <mass value="10"/>
19.                <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
20.            </inertial>
21.
22.        </link>
23.
24.        <link name="U_BASE_E2">
25.             <visual>
26.                <origin xyz="1.25 0.0 0" rpy="0.0 1.57 0.0"/>
27.                <geometry>
28.                    <cylinder radius="0.3" length="0.6"/>
29.                </geometry>
30.                <material name="GRIS">
31.                    <color rgba="0.75 0.75 0.75 0.75"/>
32.                </material>
33.             </visual>
34.             <inertial>
35.                <mass value="1"/>
36.                <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
37.             </inertial>
38.
39.        </link>
40.
41.        <link name="E2">
42.            <visual>
43.                <origin xyz="2.3 0.0 1.5" rpy="0.0 0.0 0.0"/>
44.                <geometry>
```

```
45.                     <cylinder radius="0.8" length="4"/>
46.                 </geometry>
47.                 <material name="NARANJA">
48.                     <color rgba="1 0.5 0.0 1"/>
49.                 </material>
50.             </visual>
51.             <inertial>
52.                 <mass value="3"/>
53.                 <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
54.             </inertial>
55.
56.         </link>
57.
58.         <link name="U_E2_E3">
59.                     <visual>
60.               <origin xyz="0 0.0 0" rpy="0.0 1.57 0.0"/>
61.               <geometry>
62.                   <cylinder radius="0.3" length="0.2"/>
63.               </geometry>
64.               <material name="GRIS">
65.                   <color rgba="0.75 0.75 0.75 0.75"/>
66.                </material>
67.             </visual>
68.             <inertial>
69.                 <mass value="1"/>
70.                 <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
71.             </inertial>
72.
73.         </link>
74.
75.          <link name="E3">
76.                     <visual>
77.               <origin xyz="-0.7 0.0 0" rpy="0.0 0 0.0"/>
78.               <geometry>
79.                   <cylinder radius="0.7" length="2.5"/>
80.               </geometry>
81.               <material name="NARANJA">
82.                     <color rgba="1 0.5 0.0 1"/>
83.                 </material>
84.             </visual>
85.             <inertial>
86.                 <mass value="3"/>
87.                 <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
88.             </inertial>
89.         </link>
90.
91.             <link name="U_E3_E4">
92.                     <visual>
93.               <origin xyz="0 0.0 0" rpy="0.0 1.57 0.0"/>
94.               <geometry>
95.                   <cylinder radius="0.7" length="0.5"/>
96.               </geometry>
97.               <material name="GRIS">
```

```
98.                   <color rgba="0.75 0.75 0.75 0.75"/>
99.                </material>
100.            </visual>
101.            <inertial>
102.                <mass value="1"/>
103.                <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
104.            </inertial>
105.        </link>
106.
107.            <link name="E4">
108.                    <visual>
109.              <origin xyz="-0.5 0 0" rpy="0.0 1.57 0.0"/>
110.              <geometry>
111.                  <cylinder radius="0.7" length="0.5"/>
112.              </geometry>
113.              <material name="NARANJA">
114.                    <color rgba="1 0.5 0.0 1"/>
115.                </material>
116.            </visual>
117.            <inertial>
118.                <mass value="2"/>
119.                <inertia ixx="0.4" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
120.            </inertial>
121.         </link>
122.
123.         <link name="G1">
124.            <visual>
125.                <origin xyz="-1 0.3 0" rpy="0.0 1.57 0.0"/>
126.                <geometry>
127.                    <box size="0.2 0.2 0.6"/>
128.                </geometry>
129.                <material name="GRIS">
130.                  <color rgba="0.75 0.75 0.75 0.75"/>
131.                </material>
132.            </visual>
133.            <inertial>
134.                <mass value="0.1"/>
135.                <inertia ixx="0.2" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
136.            </inertial>
137.         </link>
138.
139.         <link name="G2">
140.            <visual>
141.                <origin xyz="-1 -0.3 0" rpy="0.0 1.57 0.0"/>
142.                <geometry>
143.                    <box size="0.2 0.2 0.6"/>
144.                </geometry>
145.                <material name="GRIS">
146.                  <color rgba="0.75 0.75 0.75 0.75"/>
147.                </material>
148.            </visual>
149.            <inertial>
150.                <mass value="0.1"/>
```

```
151.                    <inertia ixx="0.2" ixy="0.0" ixz="0.0" iyy="0.2"
    iyz="0.0" izz="0.2"/>
152.                <inertial>
153.            </link>
154.
155.            <joint name="fixed" type="revolute">
156.                <parent link="WORLD"/>
157.                <child link="BASE_E1"/>
158.                <axis xyz="0 0 1"/>
159.                 <limit lower="-
   3.14" upper="3.14" effort="0.0" velocity="0.0"/>
160.
161.            </joint>
162.
163.            <joint name="E1_UE2" type="revolute">
164.                <parent link="BASE_E1"/>
165.                <child link="U_BASE_E2"/>
166.                <origin xyz="0.0 0.0 0.45" rpy="0.0 0 0.0"/>
167.                <axis xyz="1 0 0.0"/>
168.                <limit lower="-
   3.14" upper="3.14" effort="0.0" velocity="0.0"/>
169.            </joint>
170.
171.            <joint name="UE2_E2" type="fixed">
172.                <parent link="U_BASE_E2"/>
173.                <child link="E2"/>
174.            </joint>
175.
176.                <joint name="E2_UE3" type="revolute">
177.                <parent link="E2"/>
178.                <child link="U_E2_E3"/>
179.                <origin xyz="1.4 0.0 3" rpy="0.0 0 0.0"/>
180.                <axis xyz="1 0 0.0"/>
181.                <limit lower="-
   3.14" upper="3.14" effort="0.0" velocity="0.0"/>
182.            </joint>
183.
184.            <joint name="UE3_E3" type="fixed">
185.                <parent link="U_E2_E3"/>
186.                <child link="E3"/>
187.            </joint>
188.
189.                <joint name="E3_UE4" type="revolute">
190.                <parent link="E3"/>
191.                <child link="U_E3_E4"/>
192.                <origin xyz="-0.7 0.0 1.5" rpy="0.0 1.57 0.0"/>
193.                <axis xyz="1 0 0.0"/>
194.                <limit lower="-
   3.14" upper="3.14" effort="0.0" velocity="0.0"/>
195.            </joint>
196.
197.                <joint name="UE4_E4" type="fixed">
198.                <parent link="U_E3_E4"/>
199.                <child link="E4"/>
200.            </joint>
201.
```

```
202.         <joint name="E4_G1" type="fixed">
203.             <parent link="E4"/>
204.             <child link="G1"/>
205.         </joint>
206.
207.         <joint name="E4_G2" type="fixed">
208.             <parent link="E4"/>
209.             <child link="G2"/>
210.         </joint>
211.
212.             <joint name="G1_G2" type="prismatic">
213.             <parent link="G1"/>
214.             <child link="G2"/>
215.             <limit lower="-
   0.3" upper="0.4" effort="0.0" velocity="0.0"/>
216.             <axis xyz="0 1 0.0"/>
217.         </joint>
218.
219.     </robot>
```

6.2 Anexo 2: Programa de control del Niryo Ned mediante el mando de la xbox360

```
1. # !/usr/bin/env python
2.
3. from xbox360controller import Xbox360Controller
4. from pyniryo import *
5.
6. robot_ip="127.0.0.1"
7. robot = NiryoRobot(robot_ip)
8. robot.calibrate_auto()
9.
10.
11.
12.     lista_botones=['axis_l','axis_r','hat_x','hat_y','trigger_l'
   ,'trigger_r','button_a','button_b','button_x',
13.     'button_y','button_trigger_l','button_trigger_r','button_sel
   ect','button_start','button_mode',
14.     'button_thumb_l','button_thumb_r']
15.     lista_valores=[]
16.     j_robot_ant=[0,0,0,0,0,0]
17.     j_robot=[]
18.     memoria_posiciones={}
19.     lista_pos=[]
20.     estado_pinza=False
21.     pos_act=PoseObject
22.     i=0
23.     for i in range(15):
24.         lista_valores.append(0)
25.
26.     contador_hat=0
27.     contador_pos=0
28.     pos=0
29.     vel=0.01
30.     step=0
31.
32.     def on_button_pressed(button):
33.         global step
34.         boton=button.name
35.         dic_botones[boton]=1
36.
37.     def realiza_secuencia():
38.         lista_posiciones=[]
39.         cont=0
40.         tam=0
41.         if dic_botones["button_start"]==1:
42.             fin=False
43.             for key,value in memoria_posiciones.items():
44.                     lista_posiciones.append(value)
45.             tam=len(lista_posiciones)
46.             print("tam=",tam)
47.             while fin==False:
48.                 if cont==tam:
```

```
49.                     fin=True
50.                     break
51.                 robot.move_pose(lista_posiciones[cont])
52.                 cont+=1
53.
54.     def home():
55.         if dic_botones["button_select"]==1:
56.             robot.move_to_home_pose()
57.
58.     def control_pinza():
59.         global estado_pinza
60.         if dic_botones["button_a"]==1 and estado_pinza==False:
61.             robot.open_gripper()
62.         if dic_botones["button_a"]==1 and estado_pinza==True:
63.             robot.close_gripper()
64.         if estado_pinza==True:
65.             estado_pinza=False
66.         else:
67.             estado_pinza=True
68.
69.     def on_button_released(button):
70.         boton=button.name
71.         dic_botones[boton]=0
72.
73.     def gestiona_hat(hat_y):
74.         global contador_hat
75.         global step
76.         global contador_pos
77.         if dic_botones["hat_y"]== 1 and step!=1:
78.                 contador_hat+=1
79.                 if contador_hat>5:
80.                     contador_hat=0
81.                 print("EJE seleccionado:",contador_hat)
82.         if dic_botones["hat_y"]==-1 and step!=1:
83.                 contador_hat-=1
84.                 if contador_hat<0:
85.                     contador_hat=5
86.                 print("EJE seleccionado:",contador_hat)
87.         if dic_botones["hat_y"]== 1 and step==1:
88.                 contador_pos+=1
89.                 if contador_pos>=len(memoria_posiciones):
90.                     contador_pos=0
91.                 print("Elige la posicion:")
92.                 print(lista_pos[contador_pos])
93.         if dic_botones["hat_y"]==-1 and step==1:
94.                 contador_pos-=1
95.                 if contador_pos<0:
96.                     contador_pos=len(memoria_posiciones)-1
97.                 print("Elige la posicion:")
98.                 print(lista_pos[contador_pos])
99.
100.    def elige_posiciones():
101.        cont=0
102.        global step
103.        if dic_botones["button_y"]==1 and step==0:
104.            print("Elige la posicion:")
```

```
105.            lista_pos.clear()
106.            for key,value in memoria_posiciones.items():
107.                lista_pos.append(key)
108.            dic_botones["button_y"]=0
109.            step=1
110.        if dic_botones["button_y"]==1 and step==1:
111.            for key,value in memoria_posiciones.items():
112.                if cont==contador_pos:
113.                    nombre=key
114.                    break
115.                cont+=1
116.            print("Posiicon elegida:",nombre)
117.            print(memoria_posiciones[nombre])
118.            robot.move_pose(memoria_posiciones[nombre])
119.            dic_botones["button_y"]=0
120.            step=0
121.
122.    def guarda_posiciones():
123.        global pos
124.        global step
125.        global pos_act
126.        if dic_botones["button_x"]==1 and step==0:
127.            pos_act=robot.get_pose()
128.            nombre="Pose"+str(pos)
129.            memoria_posiciones[nombre]=pos_act
130.            print(memoria_posiciones)
131.            print("POSICION GUARDADA")
132.            pos+=1
133.            dic_botones["button_x"]=0
134.
135.    def gestiona_botones_trigger():
136.        global vel
137.        if dic_botones["button_trigger_l"]==1:
138.            vel-=0.1
139.            if vel<0:
140.                vel=0.01
141.        if dic_botones["button_trigger_r"]==1:
142.            vel+=0.1
143.            if vel>0.5:
144.                vel=0.5
145.
146.    def on_axis_moved(axis):
147.        global contador_hat
148.        boton=axis.name
149.        if boton=="axis_r":
150.            boton="trigger_r"
151.        if boton=="hat":
152.            dic_botones["hat_x"]=axis.x
153.            dic_botones["hat_y"]=axis.y
154.            gestiona_hat("hat_y")
155.
156.        dic_botones[boton]=axis.y
157.
158.    def on_trigger_moved(axis):
159.        boton=axis.name
160.        aux=0
```

```
161.         if boton=="trigger_r" and aux==0:
162.             boton="trigger_l"
163.             aux=1
164.         if boton=="trigger_l" and aux==0:
165.             boton="axis_r"
166.         dic_botones[boton]=axis.value
167. 
168.     def lee_botones():
169.         try:
170.             # Button A events
171.             controller.button_a.when_pressed = on_button_pressed
172.             controller.button_a.when_released = on_button_releas
   ed
173. 
174.             controller.button_b.when_pressed = on_button_pressed
175.             controller.button_b.when_released = on_button_releas
   ed
176. 
177.             controller.button_x.when_pressed = on_button_pressed
178.             controller.button_x.when_released = on_button_releas
   ed
179. 
180.             controller.button_y.when_pressed = on_button_pressed
181.             controller.button_y.when_released = on_button_releas
   ed
182. 
183.             controller.button_trigger_l.when_pressed = on_button
   _pressed
184.             controller.button_trigger_l.when_released = on_butto
   n_released
185. 
186.             controller.button_trigger_r.when_pressed = on_button
   _pressed
187.             controller.button_trigger_r.when_released = on_butto
   n_released
188. 
189.             controller.button_thumb_r.when_pressed = on_button_p
   ressed
190.             controller.button_thumb_r.when_released = on_button_
   released
191. 
192.             controller.button_thumb_l.when_pressed = on_button_p
   ressed
193.             controller.button_thumb_l.when_released = on_button_
   released
194. 
195.             controller.button_mode.when_pressed = on_button_pres
   sed
196.             controller.button_mode.when_released = on_button_rel
   eased
197. 
198.             controller.button_select.when_pressed = on_button_pr
   essed
199.             controller.button_select.when_released = on_button_r
   eleased
200. 
```

```
201.                controller.button_start.when_pressed = on_button_pre
    ssed
202.                controller.button_start.when_released = on_button_re
    leased
203.
204.                controller.hat.when_moved=on_axis_moved
205.
206.                # Left and right axis move event
207.                controller.axis_l.when_moved = on_axis_moved
208.                controller.axis_r.when_moved = on_axis_moved
209.
210.                controller.trigger_r.when_moved =on_trigger_moved
211.                controller.trigger_l.when_moved =on_trigger_moved
212.
213.            except KeyboardInterrupt:
214.                pass
215.
216.        def mueve_robot():
217.            global contador
218.            global vel
219.            global j_robot_ant
220.            global j_robot
221.            boton_pulsado=False
222.            if dic_botones["trigger_l"]!=0:
223.                j_robot=robot.get_joints()
224.                j_robot[contador_hat]=j_robot[contador_hat]-vel
225.                boton_pulsado=True
226.            if dic_botones["trigger_r"]!=0 and dic_botones["trigger_
    r"]!=-1 :
227.                j_robot=robot.get_joints()
228.                j_robot[contador_hat]=j_robot[contador_hat]+vel
229.                boton_pulsado=True
230.            j_robot_in_limits=comprueba_limites(j_robot)
231.            ejecuta_moviento=compara_lista(j_robot_ant,j_robot_in_li
    mits)
232.            if ejecuta_moviento==False and boton_pulsado==True:
233.                robot.move_joints(j_robot_in_limits)
234.
235.            j_robot_ant=j_robot_in_limits
236.
237.        def compara_lista(l1,l2):
238.            for x,y in zip(l1,l2):
239.                if x!=y:
240.                    return False
241.            return True
242.
243.        def comprueba_limites(joints):
244.            for n,j in enumerate(joints):
245.                if n==0:
246.                    if j<-2.867 :
247.                        joints[n]=-2.867
248.                    if j>2.867:
249.                        joints[n]=2.867
250.                if n==1:
251.                    if j<-1.81:
252.                        joints[n]=-1.81
```

```
253.                 if j>0.51:
254.                     joints[n]=0.51
255.             if n==2:
256.                 if j<-1.24:
257.                     joints[n]=-1.24
258.                 if j>1.47:
259.                     joints[n]=1.47
260.             if n==3:
261.                 if j<-1.99:
262.                     joints[n]=-1.99
263.                 if j>1.99:
264.                     joints[n]=1.99
265.             if n==4:
266.                 if j<-1.82:
267.                     joints[n]=-1.82
268.                 if j>0.95:
269.                     joints[n]=0.95
270.             if n==5:
271.                 if j<-2.43:
272.                     joints[n]=-2.43
273.                 if j>2.43:
274.                     joints[n]=2.43
275.
276.         return joints
277.
278.
279.     def prepara_botones():
280.         global dic_botones
281.         dic_botones=dict(zip(lista_botones,lista_valores))
282.
283.     def main():
284.         global pos_act
285.         while True:
286.             lee_botones()
287.             gestiona_botones_trigger()
288.             guarda_posiciones()
289.             elige_posiciones()
290.             control_pinza()
291.             realiza_secuencia()
292.             home()
293.             mueve_robot()
294.
295.     if __name__ == '__main__':
296.         prepara_botones()
297.         with Xbox360Controller(0, axis_threshold=0.2) as control
   ler:
298.             main()
```

6.3 Anexo 3: Programa de nodo emisor

```
7  #!/usr/bin/env python
8  import rospy
9  from std_msgs.msg import String
10
11
12 def main():
13     rospy.init_node("nodo_emisor",anonymous=True)
14     pub=rospy.Publisher("canal_1",String,queue_size=10)
15     rate=rospy.Rate(10)
16     while not rospy.is_shutdown():
17         msg="MENSAJE"
18         pub.publish(msg)
19         rate.sleep()
20
21
22
23
24 if __name__ == '__main__':
25     main()
```

6.4 Anexo 4: Programa de nodo receptor

```
1. #!/usr/bin/env python
2.
3.   import rospy
4.   from std_msgs.msg import String #Va a ser el tipo de mensaje que
    se va a intercambiar
5.
6.   def listen_msg(data):
7.       rospy.loginfo(data)
8.
9.   def main():
10.      rospy.init_node("nodo_receptor",anonymous=True)
11.      rospy.Subscriber("canal_1",String,listen_msg)
12.      rospy.spin()  #Evita salir del programa hasta que se cierre el
    nodo
13.
14.
15.  if __name__ == '__main__':
16.      main()
```

7. Bibliografía

[1] Solé, R. (2021, 15 septiembre). *Raspberry Pi: Crea proyectos DIY por muy poco dinero*. Profesional Review. https://www.profesionalreview.com/2021/07/18/que-es-raspberry-pi/

[2] Cannon, J. (2014). *Linux para Principiantes: Una Introducción al Sistema Operativo Linux y la Línea de Comandos*.

[3] Herath, D. & St-Onge, D. (2022). *Foundations of Robotics: A Multidisciplinary Approach with Python and Ros* (2022 ed.). Springer.

[4] colaboradores de Wikipedia. (2022, 7 diciembre). *Máquina virtual*. Wikipedia, la enciclopedia libre. https://es.wikipedia.org/wiki/M%C3%A1quina_virtual

[5] Ameijeiras, R. (2020, 3 noviembre). *Qué es, cómo instalar WSL2 y por qué es una gran noticia para el sector TI*. Pandora FMS - The Monitoring Blog. https://pandorafms.com/blog/es/wsl2/

[6] *Installation/Ubuntu - ROS Wiki*. http://wiki.ros.org/Installation/Ubuntu

[7] *Nodes - ROS Wiki*.

http://wiki.ros.org/Nodes

[8] *msg - ROS Wiki*.

http://wiki.ros.org/msg

[9] *Topics - ROS Wiki*.

http://wiki.ros.org/Topics

[10] *Services - ROS Wiki*.

http://wiki.ros.org/Services

[11] *Gazebo*.

https://gazebosim.org/docs/garden/getstarted

[12] *rviz - ROS Wiki*.

http://wiki.ros.org/rviz

[13] *MoveIt Tutorials — moveit_tutorials Melodic documentation*.

http://docs.ros.org/en/melodic/api/moveit_tutorials/html/index.html

[14] *Getting Ready for ROS Part 7: Describing a robot with URDF*. (2021, 25 octubre). Articulated Robotics.

https://articulatedrobotics.xyz/ready-for-ros-7-urdf/